WestVirginiaUniversity
THE RESEARCH REPOSITORY @ WVU

Graduate Theses, Dissertations, and Problem Reports

2004

Software architectural risk assessment

Ajith Reddy Guedem
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation

Guedem, Ajith Reddy, "Software architectural risk assessment" (2004). *Graduate Theses, Dissertations, and Problem Reports*. 1436.
https://researchrepository.wvu.edu/etd/1436

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

Software Architectural Risk Assessment

Ajith Reddy Guedem

Thesis submitted to the College of Engineering and Mineral Resources

at West Virginia University

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Computer Science

Katerina Goseva-Popstojanova, Ph.D., Chair

Hany H. Ammar, Ph.D.

Bojan Cukic, Ph.D.

Department of Computer Science and Electrical Engineering
Morgantown, West Virginia
2004

©2004 Ajith Reddy Guedem

ABSTRACT

Software Architecture Risk Assessment

Ajith Reddy Guedem

Risk assessment is an essential part of the software development life cycle. Performing risk analysis early in the life cycle enhances resource allocation decisions, enables us to compare alternative software architectural designs and helps in identifying high-risk components in the system. As a result, remedial actions to control and optimize the process and improve the quality of the software product can be taken. In this thesis we investigate two types of risk - reliability-based and performance-based risk. The reliability-based risk assessment takes into account the probability of the failures and the severity of failures. For the reliability-based risk analysis we use UML models of the software system, available early in the life cycle to come up with the risk factors of the scenarios and use cases. For each scenario we construct a Markov model to assess the risk factors of the scenarios and its risk distribution among the various classes of severity. Then we investigate both independent use cases and use cases with relationships, while obtaining the system-level risk factors. For use cases that include relationships we developed an algorithm that scans the entire use case diagram and aggregates the risk factors accordingly to obtain a system-level risk factor. For the performance-based risk analysis we use UML diagrams with performance related annotations, build a software execution model for each scenario and then map it to a system execution model using the deployment information. For estimating the performance-based failures of each scenario we use an asymptotic bounding analysis. The reliability-based and performance-based risk assessment methodologies are applied on various case studies.

Keywords - software architecture, reliability-based risk, Markov chain, use case relationships, performance-based risk, software execution model, system execution model, asymptotic bounding analysis

DEDICATION

I am honored to dedicate this publication to my beloved uncle, Late Shri Ranga Reddy. I express my deep love for him and wish that his soul rests in peace after his long pain and suffering. I express my deep love and affection to my parents, Mr.Sai Reddy and Mrs.Jaysree Reddy, for constantly their continuous support and motivation which to encouraged me to pursue higher education and to expand my scientific knowledge.

ACKNOWLEDGEMENTS

I express my deep gratitude to my research advisor, Dr.Katerina Goseva-Popstojanova for helping me define my research goals and for valuable guidance during this research. I thank her very much for her motivation and guidance throughout my research program. I would also like to thank the my committee members, Dr.Hany Ammar for providing me with the excellent opportunity to work with his research group and Dr.Bojan Cukic for his excellent support. I thank them all for their valuable time and contribution in the course of this study. I am also grateful to all my colleagues in the research lab. Thank you for your help and for creating a positive atmosphere that constantly motivated to expand my limits.

I gratefully acknowledge the financial support for my research provided by the NASA Office of Safety and Mission Assurance(OSMA) Software Assurance Research Program(SARP) managed through the NASA Independent Verification and Validation(IV&V) Facility in Fairmont, West Virginia.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Risk assessment provides useful means for identifying potentially troublesome software components that require careful development and allocation of more testing efforts. Risk assessment can be performed at various phases throughout the development process. Architecture models, abstract design, and implementation details describe systems using compositions of components and connectors. A component can be as simple as an object, a class, or a procedure, and as elaborate as a package of classes or procedures. Connectors can be as simple as procedure calls; they can also be as elaborate as client-server protocols, links between distributed databases, or middleware. Of course, risk assessment at the architectural level early in the life cycle is more beneficial than assessment at later development phases for several reasons. This kind of analysis highlights major architectural flaws in terms of design and identifies critical system components. Moreover risk-analysis can be applied to alternative architectures proposed early in the software life cycle and help developers choose the less risky software architecture for further design and implementation. It is well known that early detection and correction of problems is significantly less costly than detection and correction at the code level or in the later stages of life cycle.

Next, we discuss the related work on risk analysis and the major contributions of this

thesis.

1.1 Related work

This thesis presents software architectural risk analysis in terms of software reliability and performance, based on UML specifications. Software risk assessment is different from other domains and is very challenging in the sense that many parameters need to quantified and defined. The main objective of risk analysis presented in this thesis is in the lines of identifying critical components on grounds of reliability and performance. Recent evidence suggests that most faults are found in only a few of a systems components [11]. If these components can be identified early, then mitigating actions can be taken, such as, focusing the testing on high-risk components by optimally allocating testing resources [18], or redesigning components that are likely to cause failures or to be costly to maintain. Several other reliability and risk quantification ideas have been presented in [41] and [14] and [40].

We also needed to define a system model on an architectural level. A number of analytical models proposed to address software reliability are presented in [14]. According to [46] many existing software reliability models can be classified according to classification of the software systems and their maturity. For example the classification proposed in [34] is based on software life cycle such as debugging phase, validation phase or operational phase. The reliability model presented in this thesis is based on the model presented in [3]. This model considers a program flow graph of a terminating application to have a single entry and single exit and the transfer of control from component to component is described as an absorbing discrete time Markov chain(DTMC).

The reliability-based risk model presented in chapters 3 of this thesis, generalizes the software reliability model presented in [3]. First of all the models presented in our methodology

are based on UML diagrams mainly the sequence diagrams which have dynamic software behavior embedded, use case diagrams, which specify the relationships between the various use cases etc. Second instead of assuming a single exit nodes or terminating states of the software model [3], we introduce the concept of multiple failure nodes. Risk is quantified as probability of a failure times its consequence(severity/impact) of that failure. The advantage of having multiple failure nodes and mapping them with severity of failures provides a whole new dimension to the risk factor - instead of a single number risk factor is presented as a distribution among the failure severity classes. Finally the methodology presented in this thesis(chapter 3) also considers the failures of the both connectors and components rather than only component failures. This is important because the connector failures contribute and important amount of risk towards the total system risk factor.

Yacoub and Ammar [41] combine severity and complexity factors to develop heuristic risk factors for the components and connectors. Based on scenarios, they developed component dependency graph that represents components, connectors, and probabilities of component interactions. The overall system risk factor as a function of the risk factors of its constituting components and connectors is obtained using an aggregation algorithm. The reliability-based risk assessment present in chapter 3 is a lightweight methodology to perform analytical risk assessment at the architectural level based on the analysis of behavioral UML specifications, mainly use cases and sequence diagrams. This risk assessment approach is entirely analytical, in contrast with the previous works [1] and [41], which was based on simulations of execution profiles. The main advantages of the this analytical solution is is that sensitivity analysis can be performed simply by plugging different values of the parameters in the closed form solutions, which is faster and more effective than reapplying the algorithmic solutions for each set of different parameters as in [1]. Another advantage is that the development of a tool for automatic risk assessment is straightforward. We have already developed a prototype of a tool for risk assessment [44] based on the methodology presented in this paper. The tool uses Rational Rose Real Time [35] as a front end.

In risk assessment it is important to identify the components that are more prone to faults, which may manifest into failures over a period of time. Since risk analysis presented here is at an architectural-level we are interested in mapping the complexity of the component to its fault proneness. Predictive models exist that incorporate a relationship between program error measures and software complexity metrics [22]. Software complexity measures were also used for developing and executing test suites [19]. Therefore, static complexity is used to assess the quality of a software product. The level of exposure of a component is a function of its execution environment. Hence, dynamic complexity [23] evolved as a measure of complexity of the subset of code that is actually executed. Dynamic complexity used for reliability assessment purposes was discussed in [24]. Ammar et al. extended dynamic complexity definitions to incorporate concurrency complexity [1]. In addition, they used Colored Petri Nets models to measure dynamic complexity of software systems using simulation reports. Yacoub et al. define dynamic metrics that include dynamic complexity and dynamic coupling to measure the quality of software architectures [40]. Their approach was based on dynamic execution of UML state chart specification of a component and the proposed metrics were based on simulation reports.

Severity is another factor constituting the risk factor. The complexity of the component and the connectors attribute to the fault proneness or the probability of failure while severity attributes to the adversity or the consequence of those failure.A comparison of the various methodologies has been presented in [16] and a framework for estimating the severity of failure is presented in [17]. The framework combines the use of three suitable methods - Function Failure Analysis(FFA), Failure Mode and Effect Analysis(FMEA) and Fault Tree Analysis(FTA), to come up with the severity values of the components and connectors.

It is also important to consider the various relationships between the use cases are to be considered while aggregating the risk factors to compute system-level risk factor. [45] presents the various relationships between the use cases and a algorithm which automated the process of scanning the use case diagram and risk aggregations.

The second kind of risk studied in this thesis is the performance-based risk. Performance-based analysis measures risk in terms of the non-functional performance attributes of the system and is also conducted architectural level defined by UML diagrams. Several approaches have be presented, in the last few years, aimed at embedding performance(as well as other non-functional properties) information in UML software models using the UML extensions [42], [32].In work presented in [26] and [27] an extension of the UML notation to performance annotations (*pa-UML*) has been proposed to embed performance related information in UML. A framework that allows UML diagrams to be used for building performance models is presented in [20].

A different type of performance annotation on UML diagrams is carried out in [13]; the component interconnection patterns of client/server systems are investigated(to derive performance information) by use of UML class diagrams and collaboration diagrams. Dimitrov et.al [8] presents various kinds of UML extensions that are used to annotate and embed performance aspects various UML diagrams. A UML-driven framework is presented and several interesting and useful approaches of direct and expanded extensions to UML are presented such as; load-and-time weighted use case diagrams, sequence and activity diagrams with time information, state diagrams with transition probability and most importantly deployment diagrams (which map the various software components to the system hardware platform). Smith [36], [37] present the conversion of a various parameterizations of the execution graph with demand vectors, to convert it into a Execution Graph(EG). This is called the Software Execution Model [36]. A more extensive approach has been introduced in [7], where also asynchronous communication patterns and concurrent action executions have been considered.

The performance-based risk analysis presented in [5] defines the performance failure on a scenario level and quantifies risk as a combination of the probability failure and the severity of the failure. An asymptotic bounding analysis to come up with the asymptotic bounds on throughput and response time of a scenario(modelled by a software execution graph) bounding analysis is presented in [25]. This bounding analysis is light weight and gives is applicable to both open and closed queuing networks. The g

1.2 Contributions

The contributions of this thesis are summarized as follows [15]:

- The main contribution of this thesis to the methodology presented in [15] is the risk model(presented in section 3.5.2). The risk methodology presented in [15], is a light-weight methodology to perform reliability-based risk assessment at architectural level based on the analysis of UML specifications(mainly the use case and the sequence diagrams). We develop a Markov model to determine scenarios risk factors using components and connector risk factors. This model provides exact closed form solutions for the scenario risk factors. An additional advantage of the derived closed form solutions is that it provides an effective way for conducting sensitivity analysis. Thus, we simply plug different values of the parameters in the closed form solutions. Using scenario risk factors, we also derive the risk factor of each use case and the overall system risk factor. Moreover the analytical solution also helps in the automation of the risk assessment process, and the prototype of the tool that used to automate this process is presented in [44].

  The Markov model used for estimating the scenarios risk factors generalizes the existing architecture-based software reliability models in two ways. Thus, while the software reliability model presented in [3] considers only component failures, in the scenarios risk models, we account for both components and connectors failures, that is, we consider both components and connectors risk factors. Further, instead of a single failure state considered in all existing architecture-based software reliability models [14], we consider multiple failure states that represent failure modes with different severities. This approach allows us to derive the distribution of scenarios/use cases/system risk factors over different severity classes, which provide additional insights that are important for risk analysis. Thus, scenarios and use cases that have risk factors distributed among more severe classes will be more critical and deserve more attention than other scenarios and use cases. We apply domain knowledge to a frame work presented in the paper [17], to come up with severity

values of the components and connectors.

- In the paper [45], an extension to the risk assessment methodology presented in the chapter [15]is presented. The original risk-assessment methodology presented in [15] considers only independent use cases. The methodology presented in the paper [45] relaxes this assumptions of independent use cases(considering all the use cases are primitive).

  The main contribution of this thesis towards the methodology presented in [45] is the risk aggregation algorithm presented in section 4.1.4 that aggregates the risk factors of the primitive use cases into the non-primitive use cases based on the relationships. To deal with these relationships for complex systems with number of use cases, we present a general algorithm that scans the entire use case diagram, considering it as an undirected tree and come up with the risk factors of the non-primitive use cases by accounting for the primitive use cases that are directly related to those non-primitive use cases. The algorithm works in two passes. The first pass basically is a modified depth-first search tree traversal where, various kinds of use cases are differentiated and the risk factors of the primitive use cases are calculated. In the second pass the risk factors of the non-primitive use cases are calculated by aggregating the risk factors of the primitive use cases related to that use case. Finally the system-level risk factor is calculated by considering the execution probabilities of all the terminal use cases.

- In the paper [5], a methodology to calculate the performance-based risk factors is presented. A mathematical formulation of performance-based risk, as a combination of probability to violate a performance requirement and the severity of violation consequences is given. The main contribution of this thesis towards the methodology presented in [5], is the applying the asymptotic bounding analysis(presented in [25]), to come up with the bounds on response times of the scenario executions. We calculate the risk factors of the scenarios based in these bounds on the response time and workload value suitable to that scenario. We also calculate the service times of the components of the system across the various scenarios to identify the performance-critical components in the system.

1.3 Thesis Outline

The thesis is organized as follows. The current chapter presents introduction and related work. Chapter 2 presents the overview of architectural-level risk analysis, the importance and advantages of UML. Chapter 3 deals with reliability-based risk analysis, the proposed risk methodology, quantification of component/connector risk factors, overview of building/solving discrete time Markov chains(DTMC) and coming up with the various risk factors. The methodology is illustrated on the pacemaker case study(presented in 3.7). Chapter 4 deals with risk analysis with use case relationships. Chapter 5 deals with performance-based risk analysis, especially the bounding analysis which is the mail contribution of this thesis, and applies it an e-commerce case study. Finally Chapter 6 presents the conclusions and future work.

Chapter 2

Architectural-Level Risk Analysis using UML

Risk analysis can be performed at various phases throughout the software development process. But risk analysis at the architectural level is more beneficial than assessment at later phases of software life cycle for several reasons [14] [15]. Early detection and correction of problems is significantly less costly than at the code level. Architectural-level risk analysis provides an early means to identify the potentially troublesome components in software architecture. The outcomes of such analysis helps us early in the life cycle to assess the strengths and weaknesses of proposed software architectures. Moreover this also gives us a fair knowledge of the testing efforts that need to be allocated to the various components. According to the NASA-STD-8719.13A standard [30], risk is a function of the anticipated frequency of occurrence of an undesired event, the potential severity of resulting consequences and the uncertainties associated with the frequency and severity. This standard defines several types of risk such as, availability risk, acceptance risk, performance risk, cost risk, schedule risk, etc.

This thesis mainly presents our study on two types risks - reliability-based risk and

performance-based risk. The reliability-based risk takes into account the probability that the software product will fail in the operational environment and the adversity of that failure. We define risk as a combination of two factors [31]: probability of a malfunction(failure) and the consequence of that malfunction(severity). Probability of failure depends on the probability of occurrence of a fault combined with the likelihood of exercising that fault in a scenario(in which a failure will be triggered). During the early phases of the life cycle it is difficult to come up with the estimates of the probability of failure of components, hence we use quantitative factors such as complexity(for components) and coupling(for connectors) which have major impact on fault proneness according to [10]. We use dynamic metrics to come up with the probability of fault manifesting into a failure. Dynamic metrics are used to measure the dynamic behavior of the system in a given scenario based on the premise that the active components/connectors are the source of failures [40]. To determine the consequence of a failure(i.e., severity), we apply the MIL_STD 1629A Failure Mode and Effect Analysis as discussed later. Chapter 3 deals in detail with the reliability-based risk analysis. This study is based on previous work [1] and [41]. The process of risk analysis is not simple when we take into account the various relationships between the use cases. There are several relationships between the use cases and the risk aggregations process along the hierarchy the use cases is being presented in chapter 4.

The performance-based risk takes into account the probability that a software product will fail to meet its performance requirements (such as response time, throughput) in an operational environment (under some workload) and the adversity of that failure. We use several UML extensions to represent the performance attributes of the system such as annotated UML sequence diagrams, deployment diagrams and extract that information to build a system-execution model. After that a stand-alone analysis for single user workload and a contention-based analysis for a range of workloads is presented. A system execution model is built and an asymptotic bounding analysis is applied to determine the upper and lower bounds on the scenario throughput and response time for various workloads. The performance-risk factors are then calculated for workloads suitable to that scenario.

2.1 Importance and role of UML

The Unified Modelling Language(UML) [42], [43] is a widely accepted standard notation for modelling software systems and its use is continuously growing. The software development industry is embracing UML language for its various uses, starting from requirement analysis, to define software system architecture and also in the subsequent phases of software life cycle. Unified Modelling Language(UML) is the result of the unification process of earlier object oriented models and notations. The success of UML mostly relies on few elementary characteristics: different diagrams are provided(in an integrated framework) to represent the software model from different viewpoints; the language is supported by a graphical representation(easy to use), that is not far from the classical diagrams used before introducing UML(e.g., state diagrams, class diagrams, sequence diagrams); and no standard software development process is coupled to the notation, thus software designers may decide to use whatever subset of diagrams fit their application requirements and organize an application oriented software process.

Verification and validation tasks(V&V), applied to UML specifications, enable early detection of analysis and design flaws prior to implementation. V&V analysis can be categorized as static or dynamic. Static analysis helps V&V teams in reviewing the structure of UML models and generating metrics such as class size, the size of the hierarchy and static complexity measures. The complex dynamic behavior of many applications, especially real-time applications, motivates a shift in interest from traditional static analysis to dynamic analysis. Dynamic analysis is performed to analyze the behavior of objects as expected at run time.

The work presented in this thesis is mainly based on analysis done on system architecture defined through dynamic UML specifications. Risk assessment based on software-reliability presented in this thesis considers various UML diagrams. First, UML state charts are used for quantifying the fault proneness of the components, by estimating the components complexity; UML sequence diagrams for quantifying the failure probability of the connectors. Since the se-

quence diagrams inherently possess dynamic software behavior, they are also used to build the risk model based. Annotated UML use case diagrams contain information between the various use cases and they are used in the system-level risk aggregations.

Moreover UML has several extensions defined for defining various aspects of a software system. There have been a number of such extensions proposed to embed performance(as well as other non-functional attributes) parameters in UML models, most of which are discussed in the related work section 1.1. Annotated sequence diagrams are used in performance-based risk analysis(presented in chapter 5) to come up with completion times of the scenarios and to build a software execution model. Annotated deployment diagrams are used to identify the various characteristics of the hardware platform on which the software system runs and are used to map the software execution model to the system execution model. Although there needs to be more standardization of UML when it comes to some aspects of software, on the whole UML serves as an excellent tool for modelling large and complex software systems.

In the next chapter we present the reliability-based risk analysis in detail.

Chapter 3

Reliability-Based Risk Analysis

The reliability-based risk analysis identifies the potential risks in the software architecture, based on the early system specifications. The architectural specifications are the UML models that are available early in the software life cycle. The basis for the risk assessment methodology is the use case diagrams and the scenario diagrams of the system UML model. In this chapter we present a risk assessment methodology at the architectural level. Our methodology uses dynamic complexity and dynamic coupling metrics that are obtained from the UML specifications.

The risk-assessment methodology presented in this chapter considers both component and connector risk factors. We combine severity and complexity(and coupling) metrics to obtain risk factors for the components(and connectors). We combine the complexity/coupling with the severity associated with those components/connectors. A framework for estimating the severity of the is presented in our previous work [17]. Then, we develop a Markov model to estimate scenarios risk factors from the risk factors of components and connectors, by building a risk model with multiple failure states, each belonging to a severity class. Further, use cases and overall system risk factors are estimated using the scenarios risk factors. The methodology is entirely analytical which allows the automation of the process. In fact we have developed and demonstrated a

prototype of the tool(for details refer [44]. Since the solution resulting from the methodology is a closed form solution, it allows us to conduct sensitivity analysis(presented in 3.8) of the system risk factors by plugging in the different risk factor values for the component/connectors. *The main contributions of this thesis towards the reliability-based risk analysis are building the Markov or the risk model, obtaining the risk factors of the scenarios/ use cases and identification of critical components. Other steps of the methodology have been presented here for the sake of completeness.*

3.1 Overview of the proposed methodology

The methodology is a top down approach and is iterative over each level of the software architecture, starting from use case level, to scenario level and down to basic component/connector level. The process starts from a use case level, iterates through the each use case and each scenario of that use case. From a scenario level, each component and connector is analyzed and the corresponding risk factors are estimated. The component risk factor is the product of the dynamic complexity and the severity of the failure of that component, while the connector risk factor is the product of the dynamic coupling of the connector and the severity of the failure of that connector.

The risk analysis presented here is done at an architectural-level. Software architecture is defined in terms of components(states) and connectors(arcs). The components are mapped to the various components of the UML sequence diagrams and the connectors can be perceived as the medium through which the message(or control) transfer takes place. The dynamic complexity of a component is based on the state chart of that component, which comprises of the number of states of the component. Similarly, the dynamic coupling of a connector is based on the number of messages that are carried by the connector i.e. the number of messages passed from a component A to component B via the connector A-B. Severity of the component/connector is

based on classifying the impact of failure of that component/connector into four severity classes: minor, major, critical and catastrophic. This classification is based on domain knowledge applied to detailed Failure Function analysis(FFA), Failure Mode and Effect Analysis(FMEA) and Fault Tree Analysis(FTA) [17].

The scenario risk factors take into account the failure states of the components and the connectors. We build a control flow graph from the sequence diagram. The sequence diagram shows the normal transfer of the control from component to component. We have the absorbing states in the Markov chain divided into two categories: first the normal termination state called state(T), second the category of the failure states (minor, major, critical and catastrophic). Based on the component/connectors severity we decide the failure transition from that component/connector to one of the four failure states. The Markov chain is built from the scenario diagram, starting from start state(S) and then moving onto the states that first take the control of the scenario and so on. This assumption of single entry state can be easily extended to multiple entries.

The software reliability model presented in [3] considers only component failures in the scenarios risk models, but we account for both components and connectors failures, that is, we consider both components and connectors risk factors. Failure can happen during the execution period of any component or during the control transfer between the components. It is assumed that the failure of components and connectors is independent.*(Note: this assumption can be relaxed by considering higher order Markov chains).* Further, instead of a single failure state considered in all existing architecture-based software reliability models presented in [14], we consider multiple failure states that represent failure modes with different severities.

The steps we follows for the risk analysis process is presented as follows [15]:

1. *For each Use case defined for the system*

- *For each Scenario defined under that Use case*
  - *For each Component defined in the Scenario*
    - *Measure the Dynamic Complexity of the component*
    - *Assign severity based on FMEA and Hazard Analysis*
    - *Calculate the Component Risk Factor*
  - *For each Connector defined in the Scenario*
    - *Measure the Dynamic Coupling of the connector*
    - *Assign severity based on FMEA and Hazard Analysis*
    - *Calculate the Connector Risk Factor*
  - *Generate critical component/connector list*
  - *Construct Control Flow Graph and Risk model for that scenario - Markov model for that scenario*
  - *Calculate the Scenario risk factor*
- *Rank the scenarios based on their risk factor and determine critical scenarios list*
- *Calculate the Use case risk factors*

2. *Rank Use cases based on risk factors and determine the critical Use case list*
3. *Determine the critical component/connector list on a system level*
4. *Calculate the system risk Factor.*

3.2 Component/Connector Risk Factors

This work is based on the previous work [1], [40] and [41], and is presented here for completeness.

Components and connectors are the building blocks of any software architecture. Hence the architectural risk factor is again dependent on the risk factors of the components/connectors.

The assessment of component/connector risk factors is based on the dynamic UML specifications and hence these risk factors are called the dynamic risk factors. The risk factor is a product of the dynamic complexity/coupling and the severity associated with the failures of these components/connectors. Now we need to come up with the risk factors all the components and connectors in the Scenario S_x defined in each Use case U_k defined in the UML system specification.

3.2.1 Components - Dynamic Complexity

The risk factor of a component is defined as the product of the normalized dynamic complexity of the component and the severity associated with the failure of that component. The normalized dynamic complexity of a component in a scenario is obtained from the state chart of the component corresponding to that scenario. We use the McCabes cyclomatic complexity for coming up with the dynamic complexity of the component in a scenario based on the corresponding state chart. After the dynamic complexity of a component is calculated it is normalized against the sum of the dynamic complexities of all the components in that scenario. The mathematical expression for the risk factor rf_i^x of a component i in a scenario S_x is defined as:

$$rf_i^x = DOC_i^x * svt_i^x \tag{3.1}$$

where $0 \leq DOC_i^x \leq 1$ is the normalized dynamic complexity and $0 \leq svt_i^x \leq 1$ is the severity factor associated with that component.

The dynamic complexity of a component in a scenario is based on the state charts of that component. Let C_i^x denote the subset of states and let T_i^x denote the set of transitions for a component traversed in the state chart of component i in the Scenario S_x. Then the dynamic complexity of a component i in scenario S_i is obtained from the equation 3.2

$$doc_i^x = t_i^x - c_i^x + 2 \tag{3.2}$$

where $t_i^x = |T_i^x|$ and $c_i^x = |C_i^x|$(cardinal function of T_i^x and C_i^x).

The normalized dynamic complexity of a component i denoted as rf_i^x is obtained by dividing the normalized complexity of the component i obtained from the state chart by the sum of the dynamic complexities of all the components in the scenario S_x shown in the equation 3.3.

$$DOC_i^x = \frac{doc_i^x}{\sum_{k \in S_x} doc_k^x} \tag{3.3}$$

Thus the risk factor of a component takes in account the probability of the failure of the component(as reflected by the dynamic complexity) and the consequence of the failure of the component(as reflected by the severity of the component). The procedure for quantifying the severity of the failure of component is described in section 3.2.3(please refer [17] for details).

3.2.2 Connectors - Dynamic Coupling

The risk factor of a component is defined as the product of the normalized dynamic coupling of the connector and the severity of the failure of that connector. The mathematical expression for the risk factor rf_{ij}^x of the connector from a component i to j in a scenario S_x is defined as

$$rf_{ij}^x = EOC_{ij}^x * svt_{ij}^x \tag{3.4}$$

The dynamic coupling of the connector from a component i to a component j is is actually

the export object coupling from component i to j in a scenario S_x denoted as EOC_{ij}^x.

Let MT_{ij}^x denote the set of messages from a component i to a component j during the execution of a scenario S_x and let MT^x denote the set of messages exchanged between all the components that are active during the execution of scenario S_x, then EOC_{ij}^x is the ratio of number of messages sent from i to j over the total number of messages exchanged in the scenario S_x as shown in equation 3.5.

$$EOC_{ij}^x = \frac{|MT_{ij}^x|}{|MT^x|} \tag{3.5}$$

where $i, j \in S_x$ and $i \neq j$. The procedure for coming up with the severity of the failure of connectors is explained in the section 3.2.3.

3.2.3 Severity Analysis

The dynamic complexity and coupling are estimates of the fault proneness of the components and connectors, respectively. In addition to this, we also need the consequences of the potential failures of these components and connectors. Severity plays a prominent role in risk assessment and hence it is important to know the failure consequence of these components and connectors. For example, a component can have a low complexity but high severity i.e. its failure results in catastrophic loses hence it should be projected as risky. According to MIL_STD_1629A, severity considers the worst case consequence of a failure determined by the degree of injury, property damage, system damage and mission loss that could ultimately occur. Domain experts rank severity in more than one way depending on the domain of purpose [2]. Based on the analysis presented in [38] and we identify the following severity classes:

- *Catastorphic*: Failure may cause death or total system loss;
- *Critical*: Failure may cause severe injury, major property damage, major system damage, or major loss of production;
- *Marginal*: Failure may cause minor injury, minor property damage, minor system damage, or delay or minor loss of production;
- *Minor*: Failure is not serious enough to cause injury, property damage, or system damage, but will result in unscheduled maintenance or repair.

We assign severity indices of 0.25, 0.50, 0.75, and 0.95 to minor, marginal, critical, and catastrophic severity classes, respectively. The selection of values for the severity classes on a linear scale is based on the study presented in our previous work [39]. However, other values could be assigned to severity classes: for example, using the exponential scale or assuming some severity distribution (in case there is not data) or using severity-cost relationships. There is also a framework for severity analysis presented in [17] and [16].

3.3 Overview of the DTMC

The DTMC is short for Discrete Time Markov Chain. A Markov process is defined as a stochastic process whose behavior is such that probability distribution for its future development depends only on the present state and is independent of all the previous states. Discrete Time Markov Chains are a special type of Markov chains where the state space and the time are discrete, not continuous. In this analysis we are interested in a subset of the DTMC - one with absorbing states.

The software architecture is modelled by components(states) and connectors(edges/arcs). The components are mapped to the various components of the UML sequence diagrams and the

connectors can be perceived as the medium through which the message(or control) transfer takes place. The DTMC with absorbing states is very well suited for modelling the reliability of software architectures. Software architecture can be very well presented/understood by program flow graphs, control flow graphs, component dependency graphs, all of which deal with dynamics of the software system. Instead of applying the DTMC on a program control graph, which is at a low level, we apply the methodology on the system level i.e. architectural-level. This level comprises of components of the system which interact with each other according to a specific control flow. The UML sequence diagrams are an excellent representation of the various components of the system and the messages that are passed between these components (representing the control transfer or control flow). Hence these sequence diagrams can be converted to control flow graphs(or component dependency graphs). By adding the transition probabilities to the arcs of the control flow graph we build the DTMC of that sequence diagram. The details of converting the sequence diagram to a DTMC is presented in section 3.5.

One of the primary assumptions in building a DTMC (for a sequence diagram of a scenario) is that the failure of components/connectors is independent. The arcs or the edges in the Markov chain represent the transfer of control from one component (state) to another. The edges are associated with probabilities, which are basically transition probabilities. These transition probabilities are based on counting the number of messages from component to component, say, from component A to components B and C.

The terminating states of the control flow graph can be considered as the absorbing states of the DTMC. All the other states are called transient states. There is a dummy start state(S) that we add to the control flow graph to model the initial transfer of control from the S to other states. At the other end, instead of considering one absorbing state we consider several states. The absorbing states can be categorized into: one normal termination and many abnormal/failure terminations, of the control flow graph. The normal termination state represents the successful or error-free execution of the control flow graph and is represented by state(T). The failure termination states represents the failures of components/connectors. The failure states

are categorized into four classes based on the severity(or impact) associated with the failure of the component/connector (on the system).

The transition probability matrix P_{ij}^x for scenario S_x is denoted as $P_{ij}^x = [p_{ij}^x]$ where p_{ij}^x is defined as the conditional probability from a component i to a component j in a scenario S_x is defined as the conditional probability that the program control is transferred to the next component j given that it has just completed the execution in component i i.e $p_{ij}^x = n_{ij}^x/n_i^x$, where n_{ij}^x is the number of times messages have been transmitted from component i to j and n_i^x is the total number of messages from component i to all other components in that sequence diagram.

The transition probability matrix for a scenario S_x is denoted as $P_{ij}^x = [p_{ij}^x]$, where p_{ij}^x is interpreted as the conditional probability that the program will next execute component j, given that it has just completed the execution of the component i. The transition probability from component i to component j in scenario S_x is estimated as$p_{ij}^x = n_{ij}^x/n_i^x$, where n_{ij}^x is the number of times messages are transmitted from component i to component j, and $n_i^x = \sum_j n_{ij}^x$ is the total number of messages from component i to all other components that are active in the sequence diagram of the scenario S_x.

We build a transition probability matrix for each scenario S_x, $\forall$ components $i, j \in S_x$. The control flow graph is then associated with transition probabilities for the arcs . This now converts the control flow graph to a Discrete Time Markov Chain(DTMC). We then adapt a simple, well known method for calculating the steady state probabilities of the absorbing states (both normal and failure termination states). We first build a transition probability matrix P with elements p_{ij} representing the probabilities of transition from both transient-transient and transient-absorbing states s_i and apply simple mathematical results to obtain the absorbing state probabilities. These absorbing state probabilities are a direct representation of the failure probabilities (corresponding to the severity classes) of the scenario. The following section describes in detail an example of a DTMC and the procedure for calculating the probabilities.

3.4 Example DTMC with Multiple Absorbing State

An example of the transition probability matrix is given in equation 3.6 for the control flow graph given in figure 3.1. There is a single start state, s_1 and two absorbing states, T and F in the control flow graph.

$$P = \begin{array}{c} \\ s1 \\ s2 \\ s3 \\ s4 \\ T \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} s1 & s2 & s3 & s4 & T & F \end{array} \\ \left[\begin{array}{cccccc} 0 & 0.6 & 0.4 & 0 & 0 & 0 \\ 0 & 0 & 0.4 & 0.3 & 0 & 0.3 \\ 0 & 0 & 0 & 0.4 & 0.3 & 0.3 \\ 0 & 0 & 0.3 & 0 & 0.5 & 0.2 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array} \tag{3.6}$$

Figure 3.1 shows a total of six states: the first four states($S1$ to $S4$) are transient and the last two(T and F) are absorbing. The transition probability matrix shows the transition probabilities from both: transient-transient states and transient-absorbing states. The state $s1$ is the initial or start state and the control then proceeds to the states $s2$ and $s3$ with probabilities 0.6 and 0.4 respectively. Each entry p_{ij} in P matrix represents the probability of transition from state s_i to state s_j. An important to thing to note is that the sum of the elements of each row in the matrix P is equal to 1 i.e. $\sum_{i=1}^{m} p_{ij} = 1$ where m is number of columns in P matrix. Since the states T and F are absorbing states, the entries p_{TT} and p_{FF} are equal to 1 (representing the loop backs from those states to themselves).

The transition probability matrix P is partitioned into four sub matrices. This illustration shown in equation 3.7

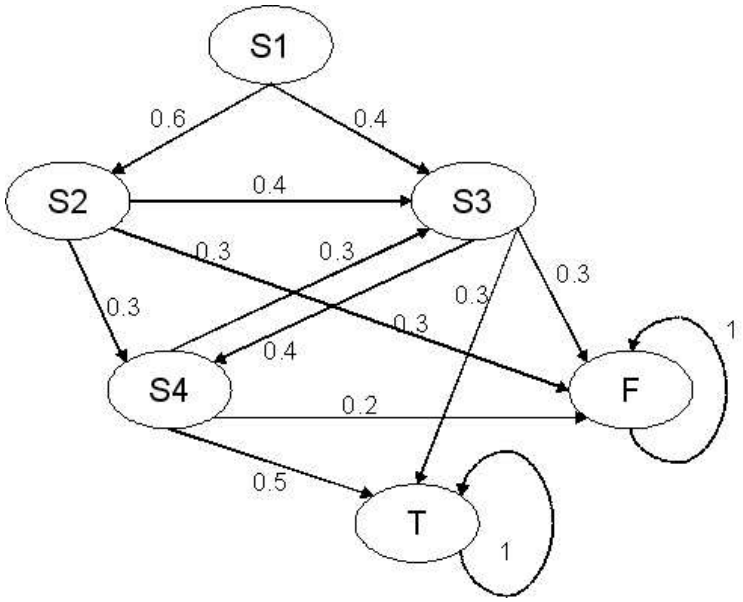

Figure 3.1: An Example Control Flow Graph.

$$P = \left[\begin{array}{cc} Q & C \\ \hline 0 & I \end{array} \right] \tag{3.7}$$

The sub matrix Q shows the transition probabilities from transient to transient states and the sub matrix C shows the transition probabilities from transient to absorbing state. The I matrix is an identity matrix with dimension equal to the number of absorbing states in the DTMC. In the above example the $Q$and C matrices are given by:

$$Q = \begin{array}{c} \\ s1 \\ s2 \\ s3 \\ s4 \end{array} \begin{array}{c} \begin{array}{cccc} s1 & s2 & s3 & s4 \end{array} \\ \begin{bmatrix} 0 & 0.6 & 0.4 & 0 \\ 0 & 0 & 0.4 & 0.3 \\ 0 & 0 & 0 & 0.4 \\ 0 & 0 & 0.3 & 0 \end{bmatrix} \end{array}$$

$$C = \begin{array}{c} \\ s1 \\ s2 \\ s3 \\ s4 \end{array} \begin{array}{c} \begin{array}{cc} T & F \end{array} \\ \begin{bmatrix} 0 & 0 \\ 0 & 0.3 \\ 0.3 & 0.3 \\ 0.5 & 0.2 \end{bmatrix} \end{array} \tag{3.8}$$

To generalize the P matrix partitioning, if a given control flow graph(CFG) has a total of n states then the P matrix is a n x n matrix. If there are m absorbing states in the CFG then the Q matrix is a square matrix with dimensions $(n-m)$ x $(n-m)$. The C matrix a matrix with dimensions $(n-m)$ x m according to the partitioning as shown in equation 3.7.

After obtaining the Q and C matrices we calculate the probability of that the control is transferred to the states T and F given the condition that the start state is S1. In the control flow graph shown in figure 3.1 the states T and F are absorbing and we are interested in the k-step transition probability for these states. The k-step transition probability of an absorbing state is defined as the probability that the DTMC is finally absorbed by that state, assuming that it starts from a start state S(in the example the start state is s_1). This absorbing state probability can be calculated by the computing the A matrix . The A matrix is defined as $A = [a_{ik}]$ where a_{ik} denotes the probability that a DTMC starting from a transient state s_i is finally absorbed in the absorbing state s_k. This A matrix can be obtained from equation 3.9.

$$A = (I - Q)^{-1} * C \tag{3.9}$$

where I is and identity matrix with the same dimensions as the Q matrix i.e. $(n-m)$ x $(n-m)$. Computing the A matrix from the Q and C matrices we have following result.

$$A = (I-Q)^{-1} * C = \begin{bmatrix} 0.4843 & 0.5157 \\ 0.4284 & 0.5716 \\ 0.5682 & 0.4318 \\ 0.6705 & 0.3295 \end{bmatrix} \tag{3.10}$$

From the expression 3.10 we have the steady state probabilities of the absorbing states T and F as 0.4843 and 0.5157. These values correspond to the elements a_{11} and a_{12} of the A matrix. It can be observed that the sum of these two values and also sum of all the rows of the A matrix is equal to 1 i.e. $\sum_{j=1}^{m} a_{ij} = 1$ where m is are the number of columns in A matrix.

3.5 Scenario Risk Factors

This section describes how to calculate the risk factors of the scenarios by building the scenario risk model. The derivation of the DTMC from the UML sequence diagram is the main step in deriving the k-step transition probability for the absorbing states steady states of the reliability model (for each sequence diagram). There are some steps involved in deriving and building the Markov model from the sequence diagram of the various scenarios. First build a control flow graph from which is a direct translation of the sequence diagram and second build the risk model for that control flow graph. Sections 3.5.1 and 3.5.2 describe the process of building control flow graphs and DTMC from the sequence diagrams.

3.5.1 Building the Control Flow Graph from Sequence Diagram

This section describes the process of building a control flow graph from the sequence diagram. The difference between the sequence diagram and the control flow graph is that the control flow graph has a single macro state for a component, while in the sequence diagram we have different active states of a component represented along the component's object life time. The convention we followed is, we named this macro state as the name of the component. Thus the states in a control flow graph represent the active components(or to be more precise the corresponding active state of that component, hidden in the representation). The arcs connecting the components(i.e. connectors) represent a transfer of control between these components.

Figure 3.2 shows a sequence diagram from the AVI scenario of a cardiac pace maker system(the cardiac pace maker case study is presented in detail in section 3.7). The sequence diagram consists of three main components - Communications Gnome(CG) which is programmed by the user to set a particular mode of system operation(in this case AVI mode), Atrial component(AR) and Ventricular component(VT)(which sense/pace the heart depending on the mode of operation). The heart shown the sequence diagram is an external actor which is sensed and paced by the pace maker system. The states of the AR and the VT components - idle, refracting, waiting, pacing are shown along the object life lines. Now this sequence diagram is converted to a control flow graph as described previously.

After obtaining the control flow graph, add the probabilities for control transfers from a component to another(represented as a number along the corresponding connector). These probabilities correspond to the transition probabilities of the P^x matrix. This gives the Discrete Time Markov Chain of the software execution behavior for that scenario.

Figure 3.3 shows the DTMC built for the AVI scenario shown in figure 3.2. It has a single entry state(state S) which is the dummy start state. An assumption here is that the control transfer between any of the states has the Markov property: *given the knowledge of*

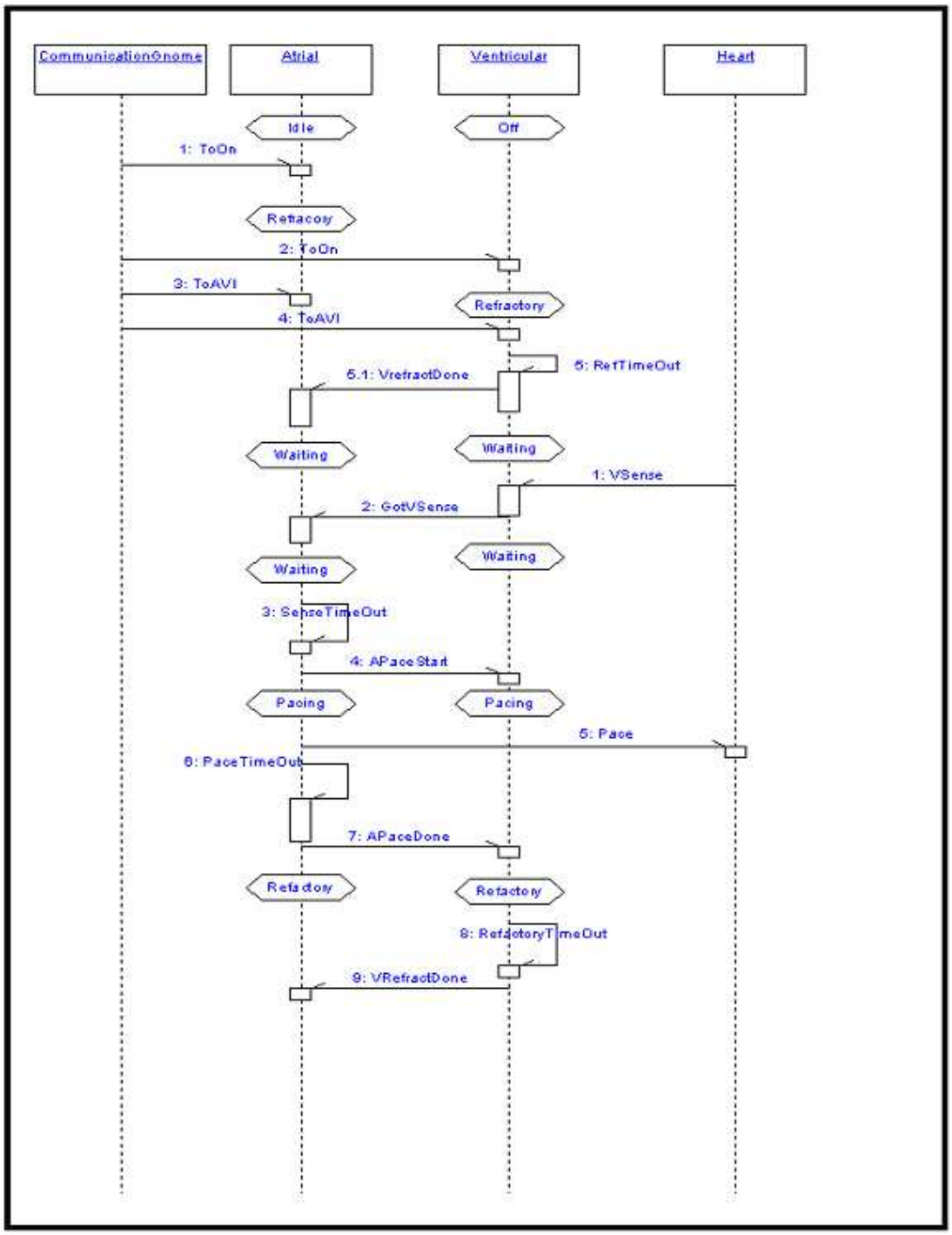

Figure 3.2: A sequence diagram in the cardiac pacemaker system.

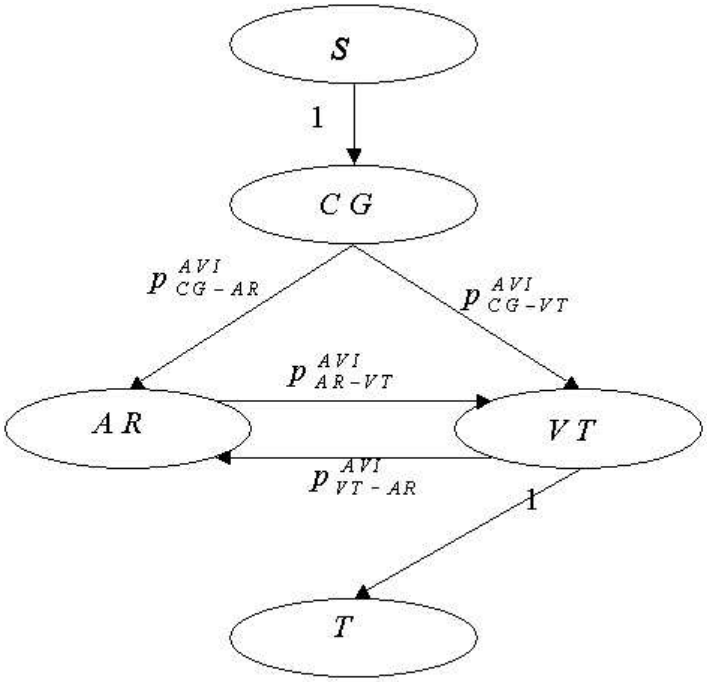

Figure 3.3: DTMC for the software execution behavior of the AVI scenario.

the component in control at any given time the future behavior of the system(or in other words the next transition) is conditionally independent of the past behavior. We now assign the basic transition probabilities of the control transfer from component to component which is denoted by the P^x for the scenario S_x. The matrix 3.11 shows the transition probability matrix P^{AVI} for the AVI scenario.

$$P^{AVI} = \begin{array}{c} \\ S \\ CG \\ AR \\ VT \\ T \end{array} \begin{array}{c} \begin{array}{ccccc} S & CG & AR & VT & T \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \tag{3.11}$$

3.5.2 Building the Risk Model

This section deals with the second step of building the risk model for the scenario. Instead of a single failure state considered in all existing architecture-based software reliability models presented in [14], we consider multiple failure states that represent failure modes with different severities. Since severity plays an important role in risk assessment, we added m failure states corresponding to the m failure modes with different severity. From our severity analysis we came up with four classes of severity. Thus we have $n + 1$ transient states(n components and the dummy start state S)and have five absorbing states(i.e. four failure states and one normal terminating state T). There could be a failure transition from a component/connector to the a failure absorbing state depending on the severity of the failure of that component/connector. If the there is no failure throughout the execution of the scenario, the control reaches the normal absorbing state(the T state). The failure states in our methodology are named after the severity associated with the failures - Minor, Marginal, Critical and Catastrophic. Now we have the transition from a component in the control flow graph to one or more failure states depending upon the severity of two kinds of failures, firstly failure of the component and secondly failure of the connectors fanning out from that component. We denote the failure of the component or the connectors fanning out of the component as a dotted line from the component to the failure state [1].

Coming back to the steps of the methodology - we modify the transition probability matrix P^x to $\overline{P^x}$. The normal transition probability p^x_{ij} between component i and j is given by:

$$(1 - rf^x_i).p^x_{ij}.(1 - rf^x_{ij}) \tag{3.12}$$

[1]Note: To make the risk model look better we do not draw a line for connector failures, instead we denote the failure of this connector by drawing a dotted line from the component fanning out that connector, to the failure state.

The normal transition probability is the conditional probability of three factors:

- $(1 - rf_i^x)$: probability that the component i does not fail;
- p_{ij}^x : probability of control transfer takes place from component i to j;
- $(1 - rf_{ij}^x)$: probability that the connector from the component i to j does not fail.

The failure transition probability of a component i is the risk factor of that component is simply the risk factor of the component i i.e. rf_i^x, while the failure transition probability of the connector between the components i and j is given by:

$$(1 - rf_i^x).p_{ij}^x.(rf_{ij}^x) \tag{3.13}$$

The failure transition probability of a connector is the conditional probability of three factors:

- $(1 - rf_i^x)$: probability that the component i does not fail;
- p_{ij}^x : probability of control transfer takes place from components i to j;
- (rf_{ij}^x) : probability that the connector from component i to j fails.

Thus the new $\overline{P^x}$ will have the dimension $n + m + 2$, because it has n components, m failure classes , S dummy start state and a T normal terminating state [2].

The notation that we follow in annotating the risk factors and transition probabilities is as follows:

[2]Note: The sum of the all the normal and failure transition probabilities from a component i is equal to one.

- $rf^{scenario-name}_{component-name}$: risk factor of the component
- $p^{scenario-name}_{connector-name}$: probability of control transfer
- $rf^{scenario-name}_{connector-name}$: risk factor of the connector

The $\overline{P^x}$ matrix can then be partitioned as follows

$$\overline{P^x} = \begin{bmatrix} Q^x & C^x \\ 0 & I \end{bmatrix} \tag{3.14}$$

Applying the equations for normal transition 3.12 on the example, the risk model for the AVI scenario the failure transition probabilities for component AR(in the AVI scenario) the normal transition probability is given by :

$$(1 - rf^{AVI}_{AR}).p^{AVI}_{AR-VT}.(1 - rf^{AVI}_{AR-VT}) \tag{3.15}$$

and the failure transition probability of the component AR is rf^{AVI}_{AR}. The failure transition probability that of the connector AR-VT, obtained by applying the equation 3.13 is given by:

$$(1 - rf^{AVI}_{AR}).p^{AVI}_{AR-VT}.(rf^{AVI}_{AR-VT}) \tag{3.16}$$

Since the severity associated with component AR and the connector $AR - VT$ is the same(Catastrophic), there is only one dotted like to represent the failure transition from the AR representing the failures of component and connector(shown in figure 3.4. Thus the failure transition probability along the dotted line is the the sum of both failure probabilities.

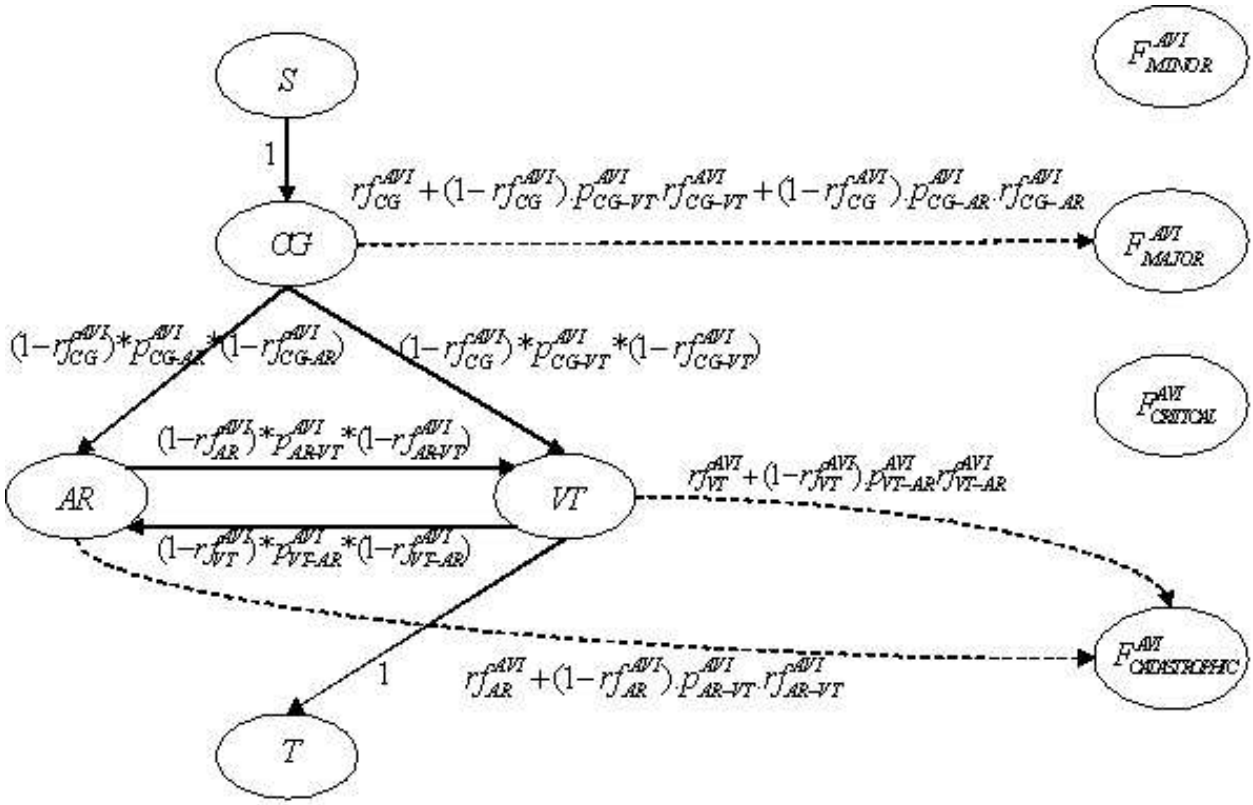

Figure 3.4: The risk model for the AVI scenario.

The process of calculating the normal and failure transitions is repeated for all the components and connectors in the control flow graph, which gives us the risk model for the AVI scenario shown in figure 3.4.

The equation 3.17 shows the $\overline{P}$ matrix for the AVI scenario [3].

[3]Note: The sum of elements along the rows of the P^{AVI} is equal to 1. This is because of the previously mentioned rule that the sum of total transition probabilities from a component should be equal to 1.

$$\overline{P^{AVI}} = \begin{array}{c} \\ S \\ CG \\ AR \\ VT \\ T \\ F_{MIN} \\ F_{MARG} \\ F_{CRIT} \\ F_{CATAS} \end{array} \begin{array}{c} \begin{array}{ccccccccc} S & CG & AR & VT & T & F_{MIN} & F_{MARG} & F_{CRIT} & F_{CATAS} \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.4998 & 0.4998 & 0 & 0 & 0.0004 & 0 & 0 \\ 0 & 0 & 0 & 0.3619 & 0 & 0 & 0 & 0 & 0.6381 \\ 0 & 0 & 0.0472 & 0 & 0.3258 & 0 & 0 & 0 & 0.6270 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array} \tag{3.17}$$

3.5.3 Solving the Markov Chain

After risk model is developed for all the scenarios(for all the sequence diagrams in each scenario) we solve inherent Markov chain of risk model. Section 3.4 already describes the procedure to solve the Discrete Time Markov Chain(DTMC) from the risk model.

The transition probability matrix $\overline{P^{AVI}}$ is partitioned into four sub matrices as shown in 3.18.

$$\overline{P^{AVI}} = \begin{bmatrix} Q^{AVI} & C^{AVI} \\ 0 & I \end{bmatrix} \tag{3.18}$$

The sub matrix Q^{AVI} has the transition probabilities from transient to transient states. The sub matrix C^{AVI} matrix has the transition probabilities from transient to absorbing state. For the AVI scenario Q^{AVI} and C^{AVI} matrices by:

$$Q^{AVI} = \begin{array}{c} \\ S \\ CG \\ AR \\ VT \end{array} \begin{array}{c} \begin{array}{cccc} S & CG & AR & VT \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0.4998 & 0.4998 \\ 0 & 0 & 0 & 0.3619 \\ 0 & 0 & 0.0472 & 0 \end{bmatrix} \end{array}$$

$$C^{AVI} = \begin{array}{c} \\ S \\ CG \\ AR \\ VT \end{array} \begin{array}{c} \begin{array}{ccccc} T & F_{MIN} & F_{MARG} & F_{CRIT} & F_{CATAS} \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.0004 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.6381 \\ 0.3258 & 0 & 0 & 0 & 0.6270 \end{bmatrix} \end{array} \tag{3.19}$$

Now we calculate the conditional probability that the program control will eventually be absorbed into one of the absorbing states given the condition that the control starts from the start state S. For this we look into the A^{AVI} matrix, the A matrix for the AVI scenario as described in the section 3.4. The success and failure probability of the scenario are as follows:

1. The probability that the control is transferred to the T state is the probability that the scenario is executed successfully.

2. Similarly the probability that the control is transferred to any of the F states : F_{minor}, $F_{marginal}$, $F_{critical}$, $F_{catastrophic}$ is the failure probability of the AVI scenario (distributed among the severity classes).

The sum of all the failure probabilities is equal to the total failure probability of the scenario or the *scenario risk factor*. According to equation 3.20, the A^{AVI} matrix is shown in 3.21.

$$A^{AVI} = (1 - Q^{AVI}).C^{AVI} \tag{3.20}$$

(a_T^{AVI}) is the probability that the AVI scenario is executed successfully and the factors $(a_{F_{MIN}}^{AVI})$, $(a_{F_{MARG}}^{AVI})$, $(a_{F_{CRIT}}^{AVI})$ and $(a_{F_{CATAS}}^{AVI})$ are the failure probabilities of the AVI scenario. These five factors correspond to the five values in the first row(because we are interested in steady state probabilities starting from start state S) of the matrix A^{AVI}. The *AVI scenario Risk factor* is shown in the equation 3.22 [4].

$$A^{AVI} = \begin{array}{c} \\ S \\ CG \\ AR \\ VT \end{array} \begin{array}{c} \begin{array}{ccccc} T & F_{MIN} & F_{MARG} & F_{CRIT} & F_{CATAS} \end{array} \\ \begin{bmatrix} 0.2256 & 0 & 0.0004 & 0 & 0.7740 \\ 0.2256 & 0 & 0.0004 & 0 & 0.7740 \\ 0.1200 & 0 & 0 & 0 & 0.8800 \\ 0.3315 & 0 & 0 & 0 & 0.6685 \end{bmatrix} \end{array} \tag{3.21}$$

$$Risk^{AVI} = (1 - a_T^{AVI}) or (a_{F_{minor}}^{AVI} + a_{F_{marginal}}^{AVI} + a_{F_{critical}}^{AVI} + a_{F_{catastrophic}}^{AVI}) \tag{3.22}$$

Thus risk factor of the AVI scenario is equal to $1 - A_T^{AVI} = 1 - 0.2256 = 0.7744$. The important advantage of this risk assessment methodology is that the failure probability/scenario risk factor the of the scenario is given as four factors, one for each class of severity. Since severity plays an important role in risk assessment, this concept of the severity specific risk factor provides vital meaning of the risk factor rather than a single number. Since the risk factor(0.7744) of the AVI scenario is distributed as 0, 0.0004, 0 and 0.7740 corresponding to minor, marginal, critical and catastrophic we know that most of the AVI scenario risk factor(99.94%) is catastrophic which is more severe than the risk factor 0.9745 distributed as 0, 0.5002, 0 and 0.4743 which has lesser catastrophic risk(only 48.67%).

[4]Note that $(a_T^{AVI} + a_{F_{MIN}}^{AVI} + a_{F_{MARG}}^{AVI} + a_{F_{CRIT}}^{AVI} + a_{F_{CATAS}}^{AVI}) = 1$.

3.6 Use case and System level Risk factors

This section deals with calculation of the use case risk factors and the system level risk factor. The risk factor rf_k of a use case U_k is obtained by averaging the risk factors of the the scenario defined under that use case. Equation 3.23 shows how to obtain the use case-risk factor, where rf_k^x is the risk factor of a scenario 'x' defined under use case 'k' and p_k^x is the execution probability of that scenario.

$$rf_k = \sum_{\forall S_x \in U_k} rf^x.p_k^x \tag{3.23}$$

Similarly the system level risk factor is obtained by averaging the risk factors of all the use cases defined in the system. Equation 3.24 shows how to obtain the system-level risk factor, where rf_k is the risk factor of use case 'k' and p_k is the execution probability of that use case.

$$rf = \sum_{\forall U_k} rf_k.p_k \tag{3.24}$$

3.7 The Cardiac Pace Maker case study

In this section we illustrate the risk assessment methodology on the Cardiac Pacemaker system.A cardiac pacemaker is an implanted device that assists cardiac functions when the underlying pathologies make the intrinsic heartbeats low. An error in the software operation of the device can cause loss of a patient's life. This is an example of a critical real-time application. We use the UML Real-Time notion to model the pacemaker. Figure 3.5 shows the components and connectors of the pacemaker in the capsule diagram. The figure also shows the input/output

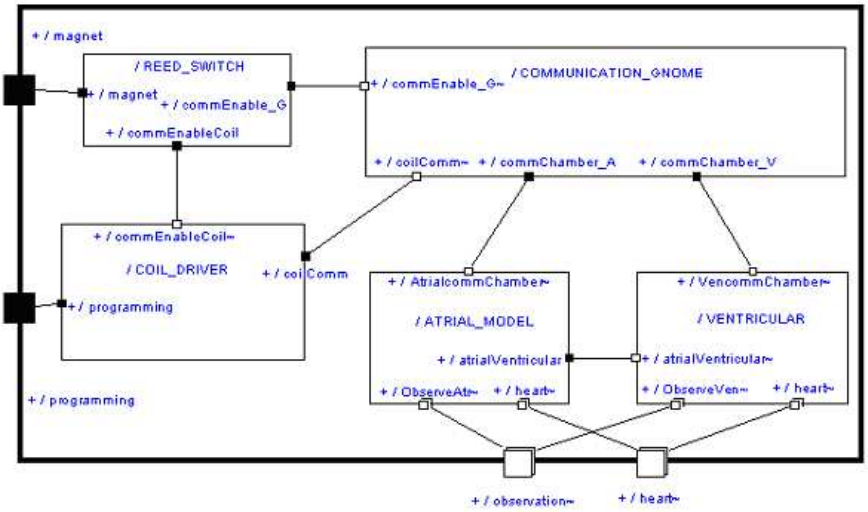

Figure 3.5: The architecture of the Cardiac Pacemaker system.

port to the Heart as an external component as well as the two input ports to the Reed Switch and the Coil Driver components. A pacemaker can be programmed to operate in one of the five operational modes depending on which part of the heart is to be sensed and which part is to be paced.

The main components of the cardiac pacemaker system are described below:

- *ReedSwitch*(RS): A magnetically activated switch that must be closed before programming the device. The switch is used to avoid accidental programming by electric noise.
- *CoilDriver*(CD): Receives/sends pulses from/to the device programmer. These pulses are counted and then interpreted as bit values of zero or one. These bits are then grouped into bytes and sent to the communication gnome. Positive and negative acknowledgments, as well as programming bits, are sent back to the programmer to confirm whether the device has been correctly programmed and the commands are validated.
- *CommunicationGnome*(CG): Receives bytes from the Coil Driver, verifies these bytes as

commands, and sends the commands to the Ventricular and Atrial models. It sends the positive and negative acknowledgments to the Coil Driver to verify command processing.

- $AtrialModel(AR)$ and $VentricularModel(AR)$: These two components are similar in operation. They both could pace the heart and/or sense the heartbeats. Once the pacemaker is programmed, the magnet is removed from the RS. The AR and VT communicate together without further intervention. Only battery decay or some medical maintenance reasons may force reprogramming.

3.7.1 The Use case model

The pacemaker runs in either a programming mode or in one of five operational modes. During programming, the programmer specifies the operation mode in which the device will work. The operation mode depends on whether the Atrial, Ventricular or both are being monitored or paced. The programmer also specifies whether the pacing is inhibited, triggered or dual. For example in the AVI operation mode, the Atrial portion of the heart is paced, the Ventricular portion of the heart is sensed(monitored) and the Atrial is only paced when a Ventricular sense does not occur (inhibited mode). The use case diagram of the pacemaker application is given in Figure 3.6. It presents the six use cases the two actors namely *doctor's programmer* and *patient's heart*. Each use case in Figure 3.6 is realized by at least one sequence diagram(i.e., scenario).

Domain experts determine probabilities of occurrence of use cases and the scenarios within each use case. This can be done in a similar fashion as the estimation of the operational profile in the field of software reliability [29]. For the pacemaker example, it follows that inhibit modes are more frequently used than the triggered mode [9]. Also, the programming mode is executed significantly less frequently than the regular usage of the pacemaker in any of its operational modes. Hence, we assume the probabilities for programming use case and five operational use cases (AVI, AAI, AAT, VVI and VVT) as given in table 3.1. Figure 3.7 shows the sequence

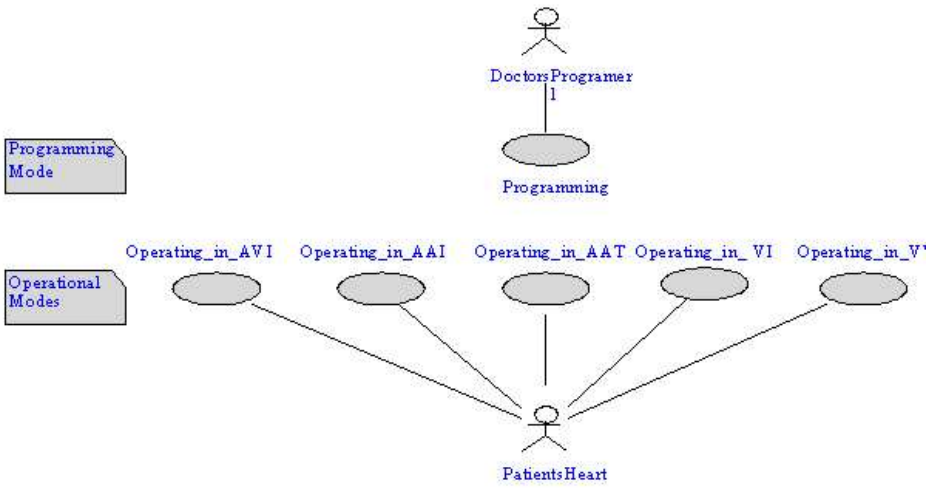

Figure 3.6: The use case model of the Cardiac Pacemaker system.

Table 3.1: The execution probability of use cases in pace maker system

Use cases	Programming	AVI	AAI	VVI	AAT	VVT
Probability	0.01	0.29	0.2	0.2	0.15	0.15

diagram of a scenario in the programming use case. In this use case the programmer interacts with the RS and CD components to input a set of 8 bits specifying an operation mode for the pacemaker. This byte is received by the CG component which in turn sets the operation mode of the AR and VT components to one of five modes(or use cases): AVI, AAI, AAT, VVI, and VVT. Figure 3.2 shows a scenario from the AVI use case in which the VT keeps sensing the heart and the AR paces the heart whenever a heart beat is not sensed. As in all scenarios, a refractory period is then in effect after every pace.

The next step is to calculate the risk factors of the components and connectors of the system architecture defined by the the basic elements(component/connectors) of the use cases and scenarios. More specifically in case of components we first calculate the normalized dynamic complexity(incase of the connectors the normalized dynamic coupling). For the normalized complexity we look into the state charts of the components under the corresponding scenario

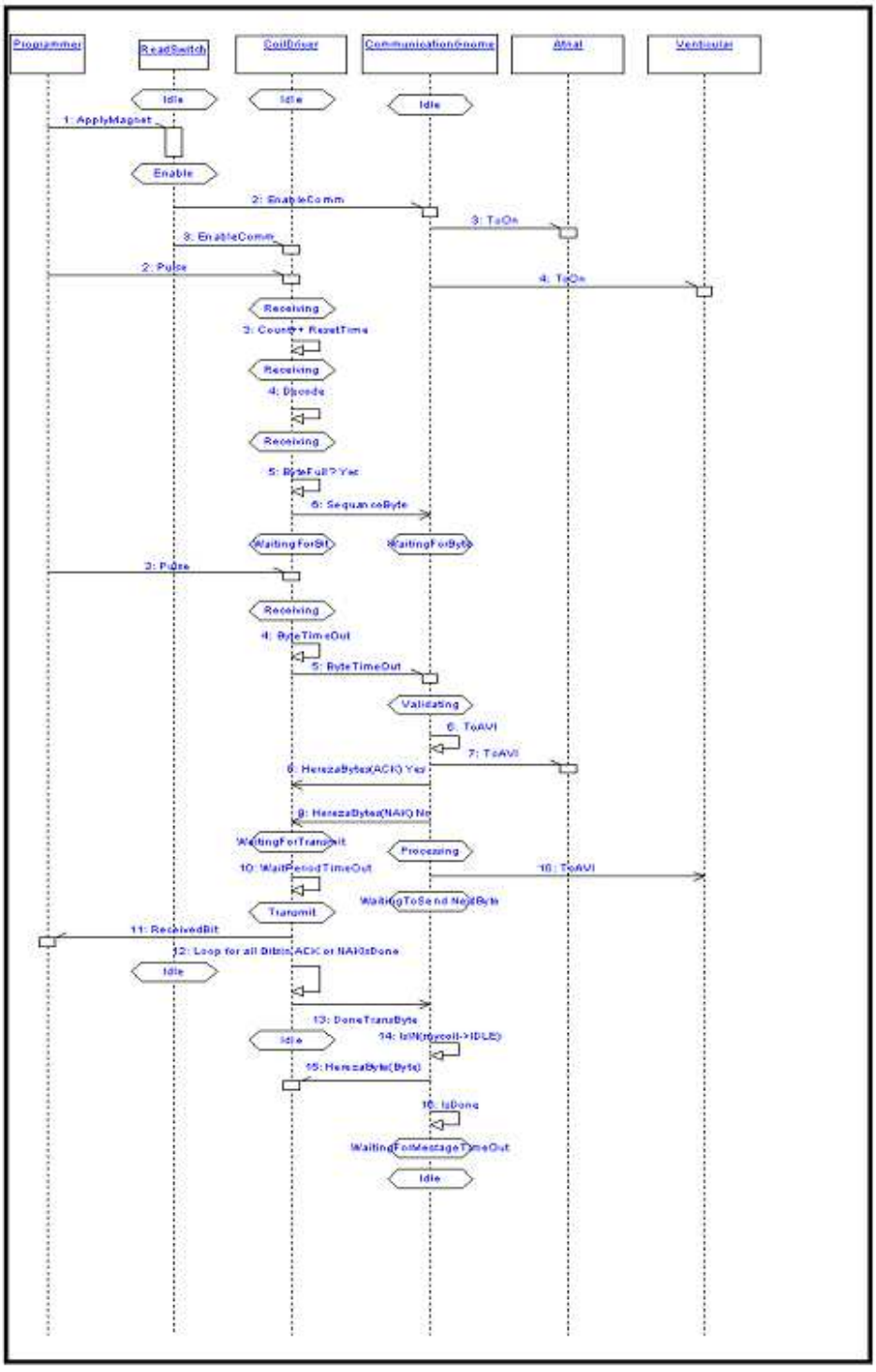

Figure 3.7: The programming scenario of the pacemaker system.

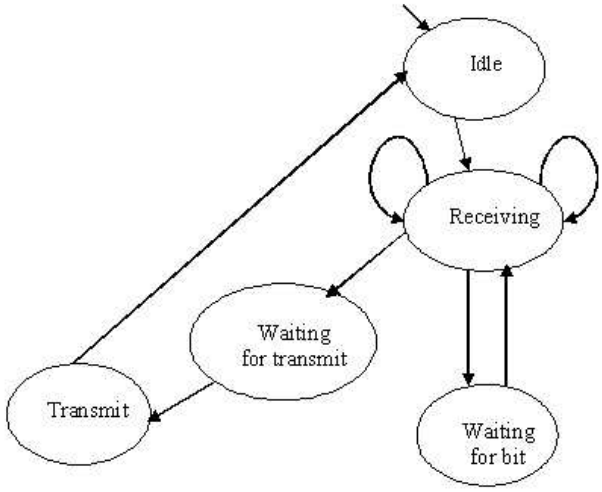

Figure 3.8: The state chart of component CD in programming scenario.

Table 3.2: The normalized dynamic complexity of all components in the Programming scenario

Component	DOC_i^x
CD	0.5
RS	0.2
CG	0.3

and come up with the McCabe's cyclomatic complexity as described in section 3.2.1. Figure 3.8 shows the state chart of the component CD under programming scenario.

Tables 3.2 and 3.3 show the normalized complexity of the components in the Programming and AVI scenarios of the pacemaker system.

Tables 3.4 and 3.5 show the dynamic coupling of the connectors between the various components in the AVI scenario. For example the value along the row RS and the column CD in table 3.4 is 0.125. This is read as dynamic coupling of the connector from RS to CD.

Then severity analysis is conducted for each component and connecter under each scenario.

Table 3.3: The normalized dynamic complexity of all components in the AVI scenario

Component	DOC_i^x
CG	0.00017
AR	0.60135
VT	0.34837

Table 3.4: The dynamic coupling of all connectors in the Programming scenario

	RS	CD	CG
RS	0	0.125	0.125
CD	0	0	0.375
CG	0	0.375	0

Table 3.5: The dynamic coupling of all connectors in the AVI scenario

	CG	AR	VT
CG	0	0.00039	0.00039
AR	0	0	0.097
VT	0	0.9	0

Figure 3.9: The severity table of all the components in the AVI scenario.

Triggered hazard	Cause of hazard	Accident	Criticality
A fault in processing command routine	Component CG misinterpreting the received bytes.	Heart is continuously triggered but device is still monitored by physician, need immediate fix or disable.	Marginal
Sensor error. Pacing hardware device malfunctioning	Component AR stays in the pacing state because it doesn't receive time out message.	Heart is always paced while patient condition requires only pacing the heart when no pulse is detected.	Catastrophic
Timer not set correctly	Component VT refract timer does not generate a timeout.	Component VT is in refractoring state, no pace is generated because it cannot sense the heart. Patient could die.	Catastrophic

Tables 3.9 and 3.10 show the results of the severity analysis applied to components and connectors respectively in the AVI scenario.

The risk factors of components/connectors is the product of their dynamic complexity/coupling and corresponding severities. After the risk factors all the components and connectors for each scenario in the system architecture are calculated, we build the scenario risk model as described in section 3.5.2 and estimate the risk factor of the all the scenario as a distribution among the failure severity classes.

3.8 Sensitivity Analysis

This section describes the sensitivity analysis that can be performed with the risk factors of the components, connectors and scenarios. The advantage with the risk assessment methodology is that we derive closed form solutions and hence a sensitivity analysis can be performed by changing the values of the parameters very easily. Compared to than the algorithmic approach

Figure 3.10: The severity table of all the connectors in the AVI scenario

Triggered hazard	Cause of hazard	Accident	Criticality
Incorrect interpretation of program bytes	Connector CG-AR transmits incorrect command, message received in error by component AR.	Incorrect operation mode and incorrect rate of pacing the heart. Device is still monitored by the physician, immediate maintenance or disabling is required.	Marginal
Incorrect interpretation of program bytes	Connector CG-VT transmits incorrect command. Message received in error by component VT.	Incorrect operation mode. Immediate maintenance or disabling is required.	Marginal
Timing mismatches between AR and VT operation.	Messages transmitted by connector AR-VT do not stop. Component AR stays in refractory state.	Failure to pace the heart.	Catastrophic
Timing mismatches between AR and VT operation.	Connector VT-AR does not transmit the messages. Component VT fails to inform component AR that the heart needs pacing.	Failure to pace the heart.	Catastrophic

presented in [41] our analytical approach is simpler, faster and more effective is performing the sensitivity analysis.

The plot in figure 3.11 shows the variation of the AVI scenario risk of the cardiac pacemaker system as a function of the risk factors of the components CG, AR and VT. As inferred from the graph the AVI scenario risk factor is more sensitive to the changes in risk factor of the component VT. This can be understood from the sequence diagram of the AVI scenario shown in figure 3.2 which shows the VT to be the most active component. We also infer that AR is also critical since it results in the smaller value of the scenario's risk factor - even for a highest risk factor assumed for the component AR, the AVI scenario risk factor does not exceed 0.85. Similarly, the plot 3.12 shows the sensitivity of the programming scenario of the cardiac pacemaker system to the component risk factors. In this case the component CG is the one that causes the largest variation in the AVI scenario risk factor(0.175 to 0.979).

We can also perform the sensitivity analysis of the system level risk factor based on the component risk factors. Plot 3.13 shows the variations in the system level risk factor by varying

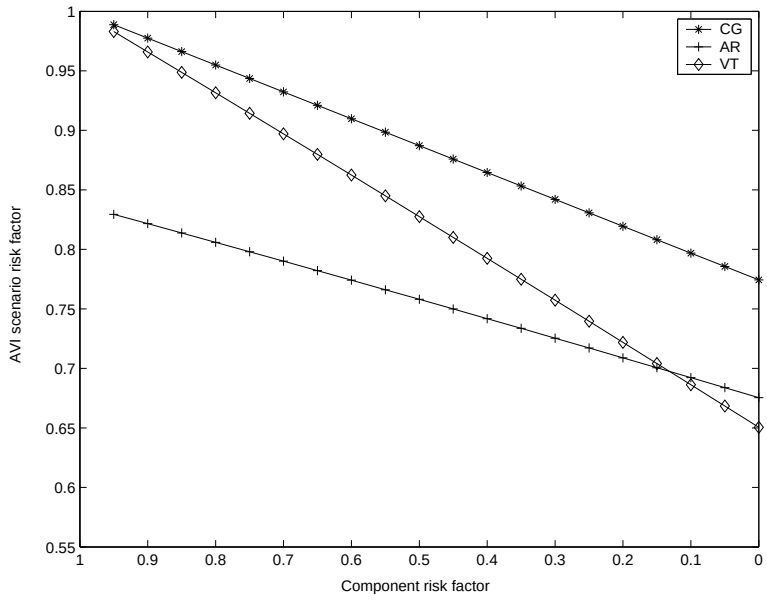

Figure 3.11: Sensitivity of the AVI scenario risk factor to the risk factors of the components.

the risk factors of the components in the cardiac pacemaker system. We can clearly infer that the components CG, VT and AR are the ones which mostly effect the system level risk factor.

Similar sensitivity analysis of the scenario and system risk factor can be performed for the connectors. Plot 3.14 shows the sensitivity of AVI scenario risk factor and plot 3.15 shows the sensitivity of the system level risk factor to the risk factors of the various connectors.

3.9 Risk factors of the Cardiac Pacemaker

The risk factors of the components/connectors, the scenarios and use cases of the cardiac pacemaker are presented in this section. Table 3.6 shows the risk factors of the various components in the various scenarios of the cardiac pacemaker system. The rows represent the scenarios and the columns represent the component. A value in the $riskm, n$ where m is the row and n is the column gives us the risk factor of the component n in the scenario m.

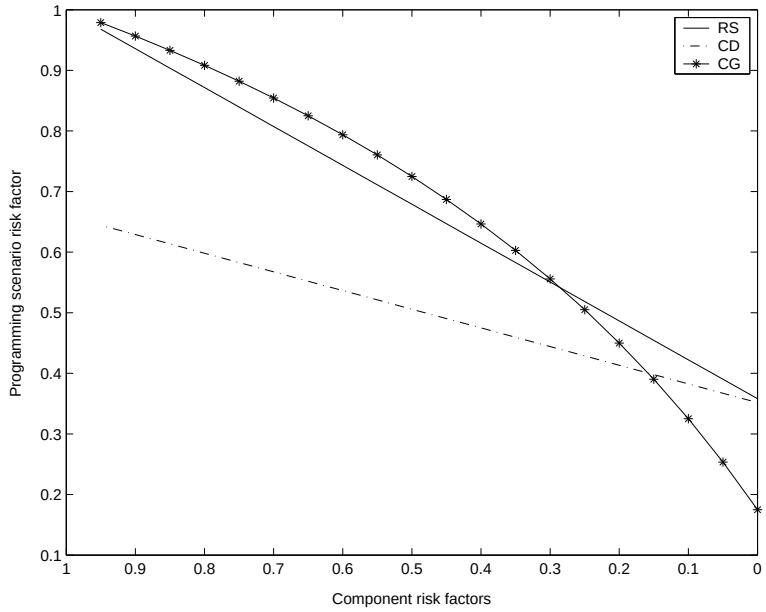

Figure 3.12: Sensitivity of the Programming scenario risk factor to the risk factors of the components.

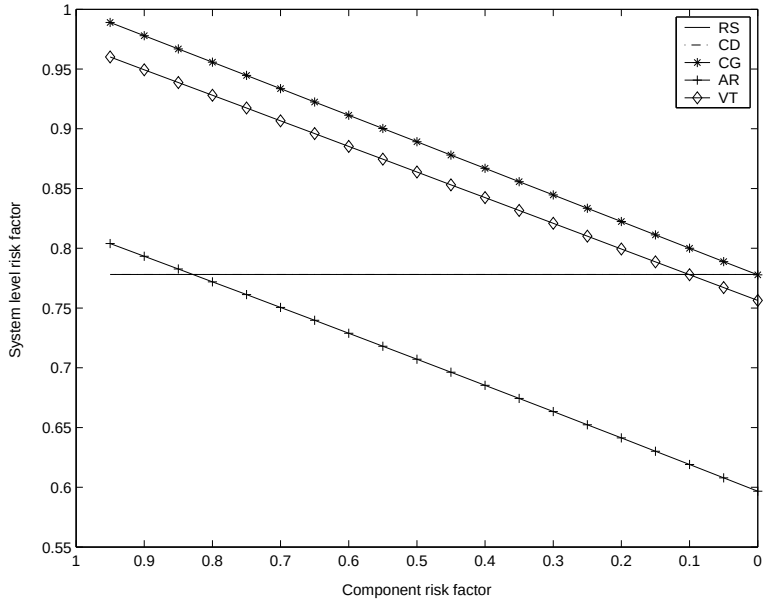

Figure 3.13: Sensitivity of the System level risk factor to the risk factors of the components.

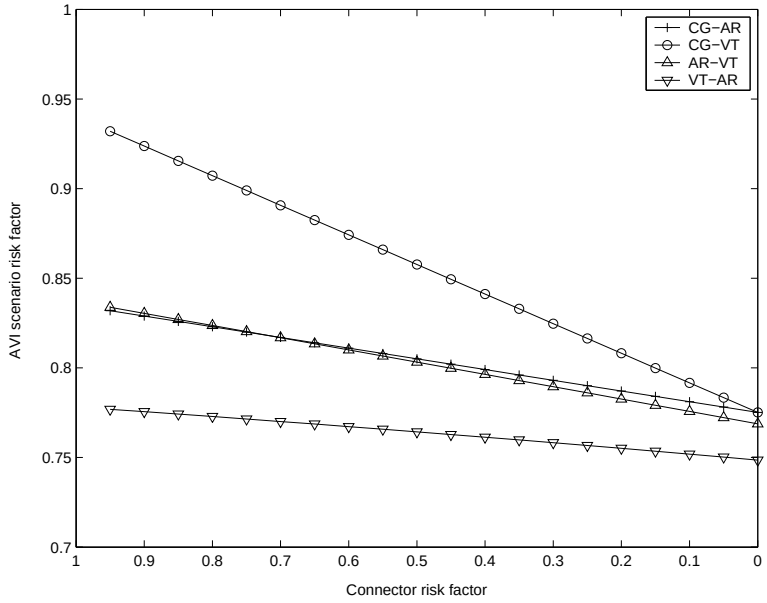

Figure 3.14: Sensitivity of the AVI scenario risk factor to the risk factors of the connectors.

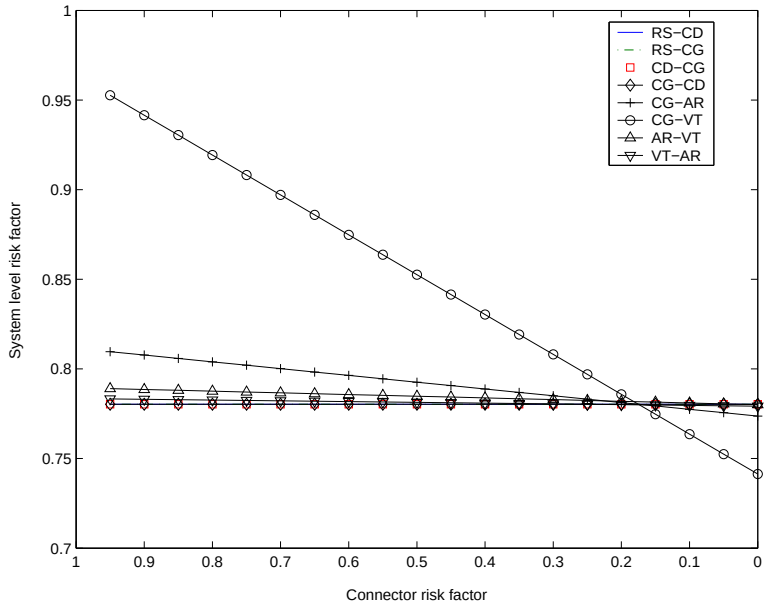

Figure 3.15: Sensitivity of the System level risk factor to the risk factors of the connectors.

Table 3.6: The risk factors of the components vs scenarios of the cardiac pacemaker.

	RS	CD	CG	AR	VT
Programming	0.05	0.125	0.25	0	0
AVI	0	0	0.00016	0.301635	0.348365
AAI	0	0	0.00045	0.94905	0
VVI	0	0	0.00045	0	0.94905
AAT	0	0	0.00025	0.949525	0
VVT	0	0	0.00025	0	0.949525

Figure 3.16 shows a 3-D bar graph of the risk factors of the component versus the scenarios of the pacemaker system. The graph is shaded according to the severity of the risk factors of the components as shown in the legend. As we can see the components VT and AR are the critical components (with darker and taller bars) across all the scenarios.

Table 3.6 shows the risk distribution of the scenarios among the four severity classes. The rows in the table represent the scenarios and the columns represent the severity classes. A value in the $riskm, n$ where m is the row and n is the column gives us the risk factor of the scenario m with the severity n. Similarly figure 3.17 shows a 3-D bar graph of the scenario risk factors versus the four severity classes. The graph is shaded according to the distribution of the scenario risk factors among the severity classes(as shown in the legend). As we can see the AVI scenario has a high catastrophic risk value. The other scenarios like the AAI, VVI, AAT and VVT have nominal marginal and catastrophic risk values.

Table 3.8 shows the distribution of the system risk factor among the four severity classes and figure 3.18 shows the cardiac pacemaker system risk distribution among the severity classes. As we can the the most part of the cardiac pacemaker system's risk falls in to the catastrophic severity class.

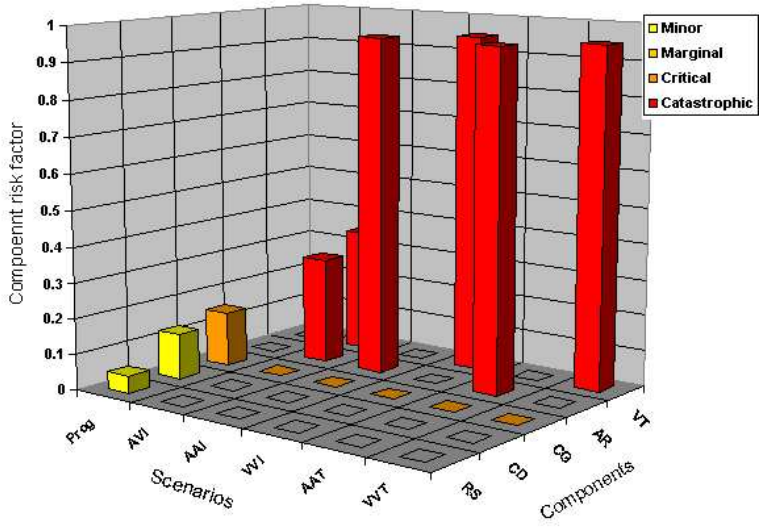

Figure 3.16: The 3-D bar graph of risk factors of the components vs scenarios of the cardiac pacemaker.

Table 3.7: The risk factors of the scenarios vs of the cardiac pacemaker.

	Minor	Marginal	Critical	Catastrophic
Programming	0.3196	0.1782	0	0
AVI	0	0.0004	0	0.774
AAI	0	0.5002	0	0.4743
VVI	0	0.5002	0	0.4743
AAT	0	0.5001	0	0.4747
VVT	0	0.5001	0	0.4747

Table 3.8: The system risk distribution of the cardiac pacemaker.

Severity Class	Minor	Marginal	Critical	Catastrophic
System Risk Factor	0.003196	0.352003	0	0.55661

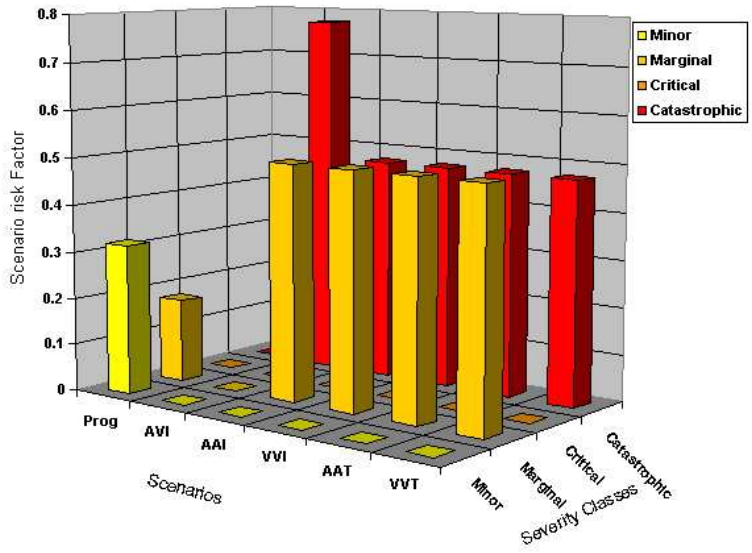

Figure 3.17: The 3-D bar graph of risk factors of the scenarios vs severity classes of the cardiac pacemaker.

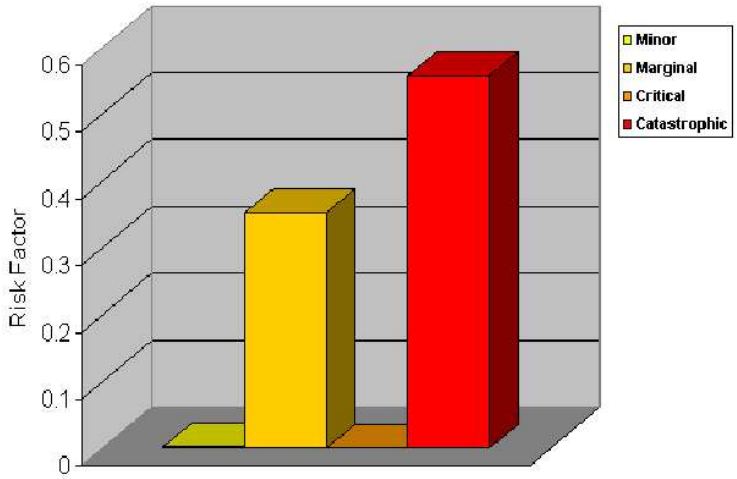

Figure 3.18: The system risk distribution of the cardiac pacemaker.

3.10 Identification of the critical components

Identification of critical components or high-risk components in a system is an important aspect of the development process of that system. This is because the set of most risky components in the system should be allocated more testing efforts than those with lesser risk. A beneficial outcome of the risk assessment methodology presented here is the ability to identify the most critical component of the system. Figure 3.16 shows a 3-D bar graph of the risk factors of the components, corresponding to their scenarios shaded according to their severity. This is one way of clearly identifying the most risky components - the component with tall bars, colored darker(as shown in the legend) in more scenarios is the most risky. Another way is to rank the components according to their risk factors and severity. Similar analysis can also be done for connectors which shows the risky connectors.

Chapter 4

Risk Analysis with Use case Relationships

The risk assessment methodology presented in the chapter 3 assumes that the use cases are independent. In this chapter we consider an extension that relaxes the assumption of independence, i.e. it takes into account the various relationships between the use cases. We present in particular, as a major contribution of this thesis to methodology described in our work [45], a traversal algorithm, which scans the use case diagram as a graph. This algorithm works in two scans of the use case diagram considering it as an undirected graph. The first pass is basically a modified depth-first traversal, which differentiates the primitive, non-primitive and the terminal use cases colors them accordingly. In the second pass it traverses the colored use case diagram and depending on the coloring scheme and the information stored in the first pass, calculates the risk factors of the non-primitive use cases. Finally the system-level risk factor is calculated by averaging the risk factors of the terminal use cases. The algorithm is described in detail in the section 4.1.4.

4.1 The various relationships between Use cases

This section is part of our risk methodology with use case relationships and is presented here for the sake of completeness. When modelling a large complex system a large amount of information is shown through the use case model. There will be relationships and commonalities between the use cases. Moreover it is expected that, during the exploration of use cases the extensions or additions to the base flow events occur as a result of interactions between the actors and the system. As the use case model is being defined, commonality through the use cases is likely to be discovered and shared. Possible extensions and additional behavior may also be uncovered and defined. Some use cases will contain behaviors that are similar in many respects. To effectively design a use case and to avoid redundant use cases there are relationships used in defining the use case model of a system.

The base use case description provides an excellent viewpoint on the overall system behaviors. The addition of alternative flow descriptions and conditional logic helps to define the variation and exception within a use case. Use case modelling provides a number of constructs that support the clear elaboration of added complexity and detail, such as “include” and “extend” relationships.

4.1.1 The Include relationship

The ≪extend≫ relationship models significant extensions and behaviors that can occur as additions to the use case model as shown in figure 4.1. The actual control transfer in this kind of extension is optional i.e. it takes place when the conditional guard is satisfied.

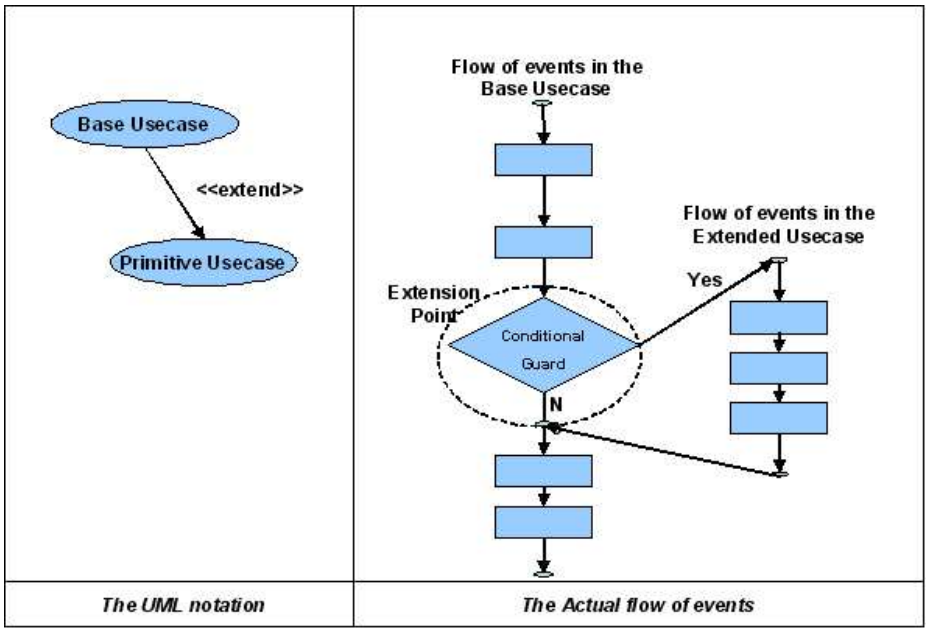

Figure 4.1: The Extends Relationship.

4.1.2 The Extend relationship

The ≪ include ≫ relationship models encapsulated behaviors that can be inserted into a use case and possible reused across multiple use cases as shown in figure 4.2. The included behavior(use case) is always exercised i.e no conditional guard is checked.

4.1.3 Use case Terminology

We use the following terminology to describe different types of the use cases and the relationships between them.

- *Primitive Use case(PUC)*: The primitive use cases are those that do not further depend on any other use cases. They can also be thought of as the terminal nodes of the use case-relation tree. The risk factors of these use cases can be calculated directly without

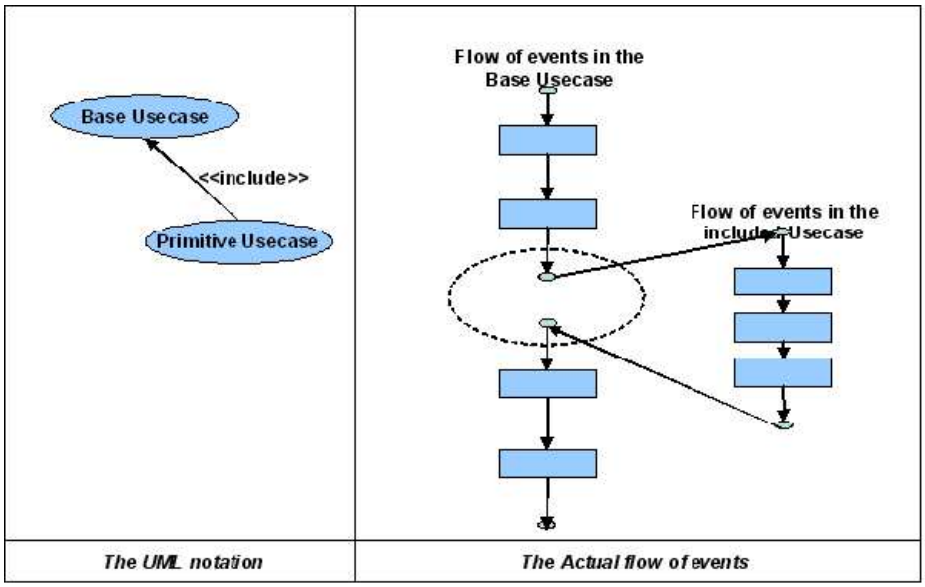

Figure 4.2: The Includes Relationship.

considering any relationships as described in the section 3.6 and in the paper [15].

- *Non-Primitive Use case(NPUC)*: The non-primitive use cases are those which are dependent on other use cases. They can be dependent on both primitive use cases and also other non-primitive use cases. The risk factor for such a use case cannot be calculated directly, because it depends on the risk factors of the use case that are related to this non-primitive use case. The algorithm described in 4.1.4 describes how to calculate such risk factors. The base use cases described previously is non-primitive.

- *Terminal Use cases(TUC):* The terminal use cases are those which are directly associated with the actor. The terminal use cases can be both primitive or non-primitive. The risk factors for such use cases are calculated using the algorithm described in 4.1.4.

The figure 4.3 shows an examples of the various kinds of use cases and relationships between them. An example of the *ll*extends*gg* relationship is shown in the left part of the figure and the *ll*includes*gg* relationship is shown in the right part of the figure.

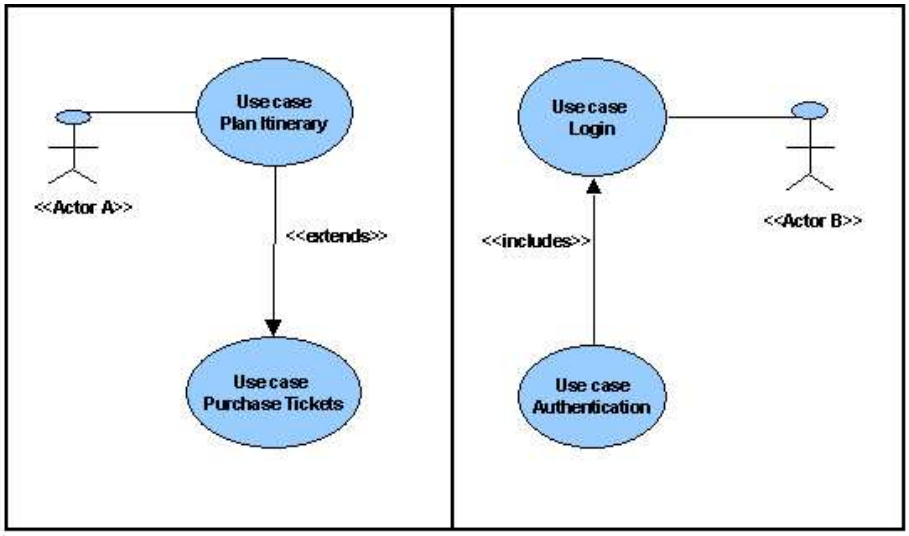

Figure 4.3: An example use case diagram showing the hierarchy of dependent use cases.

Considering the ≪extends≫ relationship in the figure 4.3 we have the use case "Plan Itinerary" associated directly with the actor A, hence this is a terminal use case(TUC). The use case "Purchase Tickets" is extended by the "Plan Itinerary" use case hence, the "Plan Itinerary" use case is a non-primitive use case(NPUC) and the "Purchase Tickets" use case is a primitive use case(PUC) as it is not further related to other use cases. Now focussing on the llincludesgg relationship we have the "Login" use case directly associated with the actor B, hence it is a terminal use case(TUC). The use case "Authentication" is included in the use case "Login", hence the "Authentication" use case is a non-primitive use case and the "Login" use case is primitive as it is not further related to any other use case.

4.1.4 The Algorithm

This section describes the algorithm to come up with the risk factors of the use cases according to their relationships, given an annotated use case diagram. Before going into the details let us look at how the algorithm works on the use case diagram shown in figure 4.3. In general, to deal with the relationships we visualize the use case diagram as a graph. First we calculate the risk factors

of the primitive use cases(such as "Purchase Tickets" and the "Authentication" use cases) and then aggregate these risk factors into the non-primitive use cases(such as "Plan Itinerary" and "Login" use cases) that are directly related to the primitive use cases. This process is repeated until we estimate risk factors of all the terminal use cases(the "Plan Itinerary" and "Login" use cases). After that calculate the system-level risk factor by averaging the risk factors of the TUC with the corresponding execution probabilities. The process of calculating the risk factor for a primitive use case(C) is based on the methodology presented in the section our previous work [15].

Risk factor aggregations are done depending upon the relationships between the use cases and the amount of detail about the use cases. In the case of ≪includes≫ relationship the probability P(i.e the probability that the PUC will be executed in the DTMC of NPUC) is equal to one and incase of ≪extends≫ relationship the P is equal to the execution probability assigned to that PUC.

For the example use case diagram in 4.3 the risk factor calculated from the DTMC of the "Authentication" use case, its risk factor in are incorporated into the DTMC of the "Login" use case always i.e. with $P = 1$ as the relationship is llincludesgg. In case of the "Purchase Tickets" use case , its risk factor is included in the "Plan Itinerary" use case with a probability $0 < P < 1$, which is is equal to the execution probability o"Purchase Tickets" use case as specified, because the relation ship between use cases A and B is ≪extends≫. Finally we average the risk factors of use cases A and C with the system-level execution probabilities(as specified), to come up with the system-level risk factor.

This process might look simple for the examples presented in the figure 4.3 but in practise the use case diagrams are huge and complex. Hence we realized the need for a general algorithm which scans the use case diagram, identifies the use cases according the relationships between them, identifies the hierarchy and then aggregates the risk factors of the use cases according to the relationships.

The algorithm presented here needs a properly annotated use case diagram(graph) as input. Annotations of the use case include - proper nomenclature of the actors and the use cases and the probabilities across the arcs for the relationships between the use cases. The following algorithm works in two passes(or scans) for a given use case diagram. In the first pass the algorithm traverses the use case diagram using a depth-first priority. The main outcome of the first scan is to identify and color the use cases according to the relationships and hierarchy. This helps in the risk aggregations during the second pass. There could be special use cases.

By the end of the first pass/scan we have the use cases(nodes) of use case diagram(graph) colored with four colors - red for actors, white for primitive use cases, black for non-primitive use cases and gray for use cases that are associated directly with the actors. There are also combinations of two colors: such as $(gray + black)$ or $(gray + white)$ if a particular use case satisfies the criteria for both the conditions. We also build the Successor_RiskMatrix[] list for each node in the graph, which has the same dimension as the Adjacent[] matrix, that stores the risk factors of the all the child nodes of each node as soon as they are calculated. By the end of first pass we also calculate the risk factors of the primitive use cases and their corresponding entries in the Successor_RiskMatrix[] of the parent nodes. [1]

In the second pass of the algorithm we scan the colored use case diagram - we start from an actor, and then look into the Successor_RiskMatrix[] of each node. If all the values in this matrix are calculated(i.e. no value in the matrix is -1) then we directly calculate the risk factor of that node by calling the Risk-Factor() routine, otherwise we proceed in the direction of the node which has a -1 entry in its parent/predecessor Successor_RiskMatrix[] and look into its Successor_RiskMatrix[] recursively. This process is repeated until the Successor_RiskMatrix[] of the all gray colored nodes are calculated. Finally the System-Risk() routine is called, which calculates the system risk factor.

The risk assessment algorithm is described as follows [5]:

[1]Initially all the entries of the Successor_RiskMatrix[] for each node are initialized to -1

Main()

Initialize all the Succcesor_RiskMatrix[] of the all nodes to -1;

The Succcesor_RiskMatrix[] of each node is of the same dimension as the Adjacent[] matrix of that node and it stores the risk factors of the child nodes(corresponding to the child of the Adjacent[] matrix) or -1 in case the risk factor of the child node is not calculated. -1 is used because it is an invalid risk factor which acts as an indicator that the risk factor of a particular node is not calculated yet

Initialize Risk_Factor value of all nodes to -1;

-1 is used for the same reason and the Risk_ Factor(node) stores the risk factor of the node

First-Pass();

In the first pass all the nodes of the use case diagram are traversed and each node is colored according to the coloring scheme. Also by the end of the first pass risk factors of all the primitive nodes is calculated and stored in the corresponding places in the Succesor_RiskMatrix[].

Reset Adjacent[] matrices of all the nodes as they have been decremented in the first pass;

Second-Pass();

In the second pass, the risk factors of all the non-primitive nodes are calculated after which the system-level risk factor are calculated

End Main();

First-Pass()

1. **For each a ∈ Actors[]**

   For each actor in the use case diagram

   (a) **color(a)=red;**

   Color the actor red

   (b) **Predecessor[a]=Nill;**

   Set its predecessor value to Null

   (c) ***Direct-Actor(a);***

   Call the routine Direct-Actor with the corresponding actor as the parameter

   (d) **Actors[]=Actors[]- a;**

   Remove that actor from Actors[]so that it the traversal does not start from that actor again

2. **Return**

3. **End First-Pass()**

Direct-Actor(a)

1. **For each u ∈ Adjacent[a]**

   For each node directly connected to the actor a

   (a) **color(u)=gray;**

   Color it gray

(b) **Predecessor[u]=a;**

Set its predecessor value to the related actor

(c) **Adjacent[a] = Adjacent[a] - u;**

Remove that node from the Adjacent[] matrix

(d) ***DFS-Visit(u);***

Repeat these steps until all the nodes connected to that actor are traversed

2. **Return**

3. **End Direct-Actor()**

DFS-Visit(u)

1. **If(color(u)==red) return;**

If we reach the actor which is colored red then return

2. **If(Adjacent[u] is not null)**

While the Adjacent[] of the node u is not null

(a) **For each v ∈ Adjacent[u]**

For each such node

i. **If color(v)==nil Predecessor[v]=u ; color(v)=black;**

Color it black for mark it as non-primitive node

ii. **If(color(v)==gray Predecessor[v]=u; color[v]=black + gray;**

Color it(black + gray) to mark it as non-primitive node if it is already gray

iii. **Adjacent[u]=Adjacent[u] - v;**

Remove that node from the Adjacent[] so that we do not go back in that direction

iv. ***DFS-Visit(v);***

Recursive call of the routine with node v as parameter

3. **Else**

   If there are no more child nodes

   (a) **If(color(u)==gray color(u)=white + gray;**

   Color that node white + gray is already gray

   (b) **Else color(u])=white;**

   (c) **Calculate the primitive use case risk factor and store it in Risk_Factor[u];**

   (d) **Successor_RiskMatrix[predecessor(u)]= Risk_Factor(u);**

   Store it in the Successor_RiskMatrix of the parent

   (e) ***DFS-Visit(predecessor[u]);***

   Go a the parent of the node u and traverse along the other nodes

4. **Return;**

5. **End DFS-Visit()**

Second-Pass()

1. **For each u ∈ Adjacent[color(node)==red]**

   For each node u directly connected to the actor in the use case diagram

2. ***DFS-Scan(u);***

   Call the routine DFS-Scan with parameter u

3. ***System-Risk();***

   Call the system risk factor

4. **Return;**

5. **End Second-Pass();**

DFS-Scan(u)

Does a depth first scan each time for the node whose risk factor is not calculated

1. **If(color(u)==red) return;**

   If we reach the actor which is colored red then return

2. **If(Risk_Factor(u)==-1)**

   If the risk factor of the use case is not calculated yet

   (a) **For x ∈ Successor_RiskMatrix[](u)==-1**

   For each nodal entry 'x'==-1 in the Successor_RiskMatrix of node u

   (b) ***DFS-Scan(x);***

   Initiate a recursive DFS-Scan with that node as parameter

3. **Else**

   (a) ***Risk-Factor(u);***

   Call the routine Risk-Factor with node u as parameter

   (b) ***DFS-Scan(predecessor(u));***

   Call the routine recursively with parent of node u

4. **Return;**

5. **End DFS-Scan()**

Risk-Factor(u)

1. **Read the probabilities of all the arcs along the nodes Adjacent[u];**

   Remember all the Adjacent nodes have been reset before second pass

2. **Read the values of the Successor_RiskMatrix[](u);**
3. **Build/solve the Markov chain with u as the start node and considering only those nodes directly connected to u;**
4. **Successor_RiskMatrix [predecessor(u)]= Risk_Factor(u);**
5. **Return;**
6. **End Risk-Factor(u);**

System-Risk()

1. **For all the(gray + white) and(gray + black) colored nodes calculate the product of execution probability and the risk factor;**
2. **System_Risk_Factor = Sum(products);**
3. **Return;**
4. **End System-Risk()**

4.2 A Motivating case study

The case study we use to illustrate our work is a large command and control system that is a real time system used in a life-critical, mission-critical application. The modelling of the system was performed using Rational Rose Realtime case tool. We will concentrate on the Thermal Control part of the system, which has a hierarchical architecture.

Figure 4.4 shows the use case diagram and all the relationships among the use cases and the actors. It is a rather complex system with operations setting controller, fault recovery

procedures and pump control functionalities. The system is responsible for providing overall management of pumps as well as performing the necessary monitoring and response to sensors data. Also, it is responsible for performing automated startup and controlling thermal system reconfigurations.

During each execution cycle, a check is performed for incoming commands. Received commands are validated in the same execution cycle. Mode change commands, which will reconfigure the internal thermal system are also accepted from other components of thermal system to compensate for system component failures or coolant leaks. A failure recovery system detects failure conditions and performs recovery operations in response to the detected failures. Failure conditions include combinations of pump failures and shutoff-valve failures.

4.2.1 Scenario Risk Factors

We use an analytical modelling approach to derive the risk factor of each scenario. For this purpose we generalize the state-based modelling approach previously used for architecture-based software reliability estimation [14] in chapter 3. In the scenario risk model we account for both component and connector failures, that is, consider both component and connector risk factors. In addition, instead of a single failure state for the scenario, we consider multiple failure states that represent failure modes with different severity.

This approach allows us to derive not only the overall scenario risk factor, but also its distribution over different severity classes, which provides additional insights important for risk analysis. For example, two scenarios may have close values of scenarios risk factors with significantly different distributions among severity classes. It can then be inferred that the scenario with a risk factor distributed among more severe failure classes (e.g., critical and catastrophic), deserves more attention than the other scenario. The scenario risk model is developed in two steps as described in sections 3.5.1 and 3.5.2.

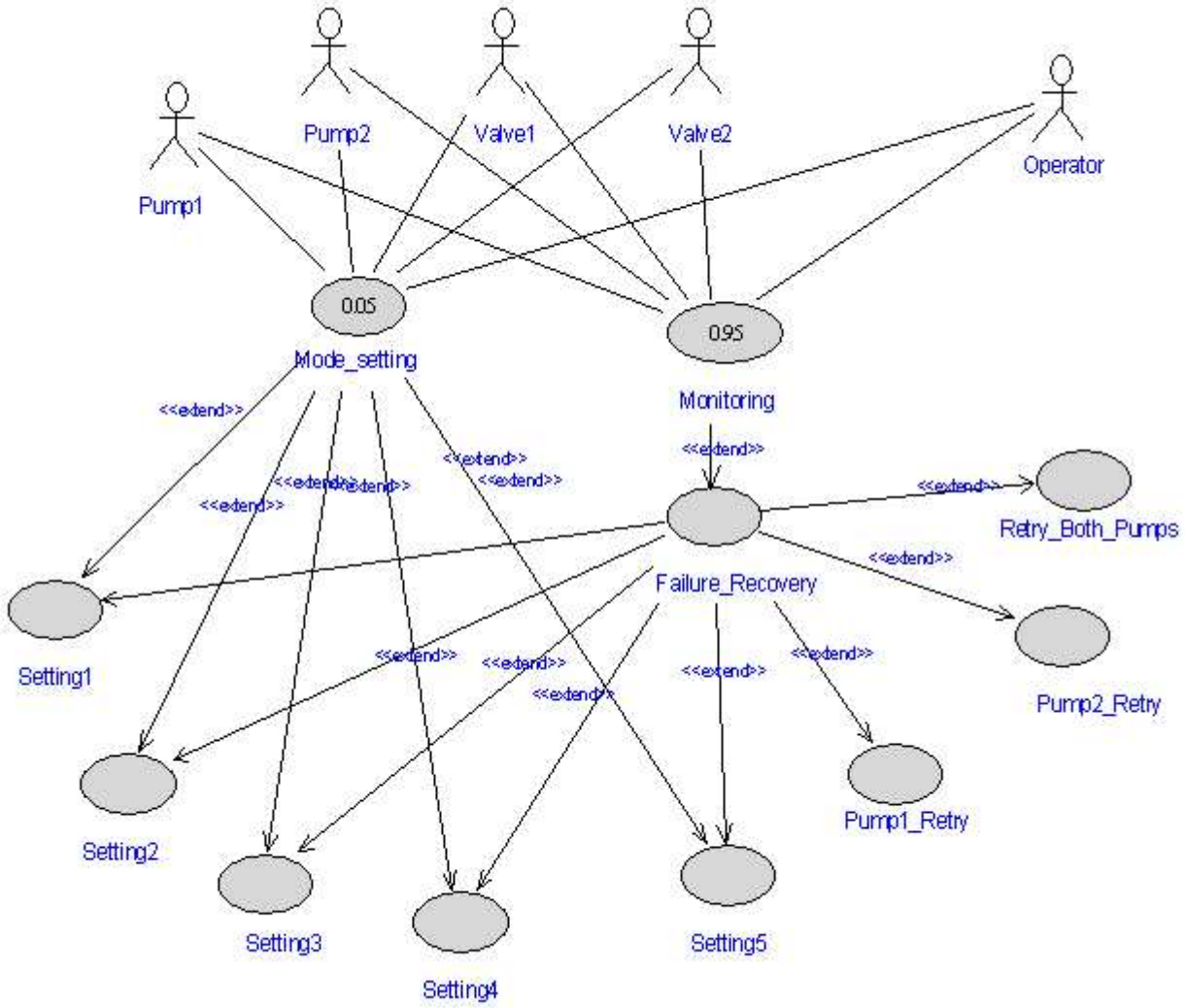

Figure 4.4: The Use case diagram of Thermal Control.

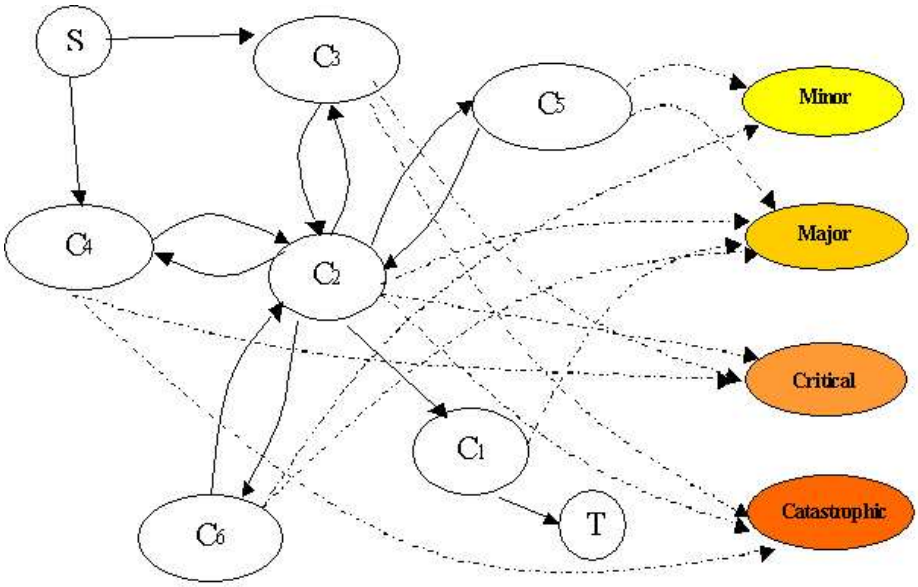

Figure 4.5: The Risk model for Both Pump Retry scenario.

Applying the methodology described in [15] the risk factor of the Both Pumps Retry scenario is calculated to be 0.76. This risk factor is distributed among Marginal, Critical and Catastrophic severity classes as 0.11, 0.0703 and 0.5802 respectively. The figure 4.5 shows the risk model built for the bothpumpretry scenario shown the use case diagram 4.4.

4.2.2 Use case Risk Factors

To calculate the risk factors of the use cases according to the relationships between them we apply the algorithm described in section 4.1.4 on the use case diagram shown in figure 4.4. In this case study we are considering only the ≪extend≫ relationship(shown in figure 4.4). As per the algorithm the risk factors of primitive use cases - the Setting use cases(Setting1,Setting2...) and the Pump_Retry use cases(Pump1_Retry,Pump2_Retry...) are calculated first. Then the DTMCs for the Failure_Recovery and Mode_Setting use cases are built by incorporating the extended use cases. Since the Failure_Recovery use case is extended further by the Monitoring use case, the DTMC of the Monitoring use case incorporates Failure_Recovery. From the domain

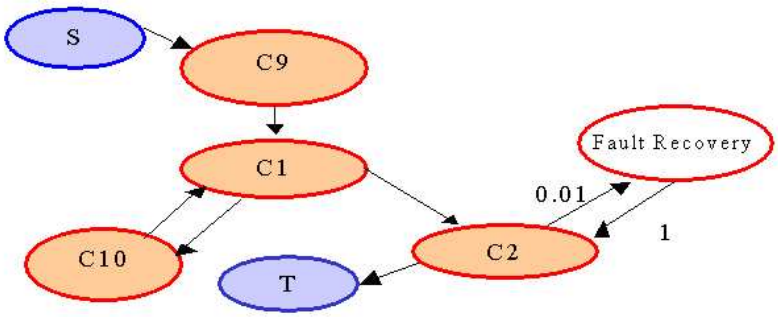

Figure 4.6: The DTMC for the Monitoring use case.

knowledge, we know that the Failure Recovery use case is extended with probability of 0.01. Thus the DTMC for the monitoring use case is is generated accordingly, as shown in figure 4.6.

Finally the risk factors of the terminal use cases Failure_Recovery and Monitoring are averaged with the execution probabilities 0.05 and 0.95 respectively to obtain the system-level risk factor. Figure 4.7 shows a 3-D bar graph of the distribution of risk factors of various use cases 'extended' by the Failure_Recovery use cases.

Figure 4.8 shows a similar graph for the various use cases 'extended' by the Mode_Setting use case. The bars are colored according to the severity of the risk factors. A taller and darker bar represents a risk value with high severity. As shown in the figure the much of the risk factors of the primitive use cases of the non-primitive Mode_Setting use case fall into the catastrophic severity range. Moreover the risk factors values of the use cases are higher than those of the Failure_Recovery use case. It can be clearly inferred that the use cases need thorough revision and more testing efforts.

In figure 4.9, the risk factors of the monitoring and mode setting use cases are presented. The bars represent the total risk factors of the use cases along with the distribution of risk factors among the severity factors. Monitoring use case is clearly the riskier than mode setting. This is because the execution probability of the Monitoring use case is high. Had we not considered this

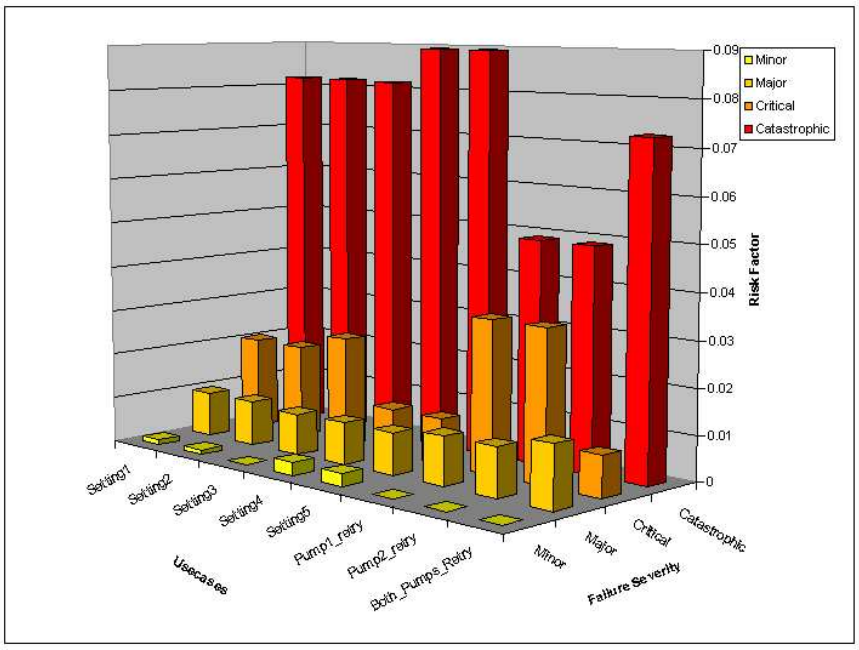

Figure 4.7: The risk distribution of the use cases for the Failure_Recovery use case.

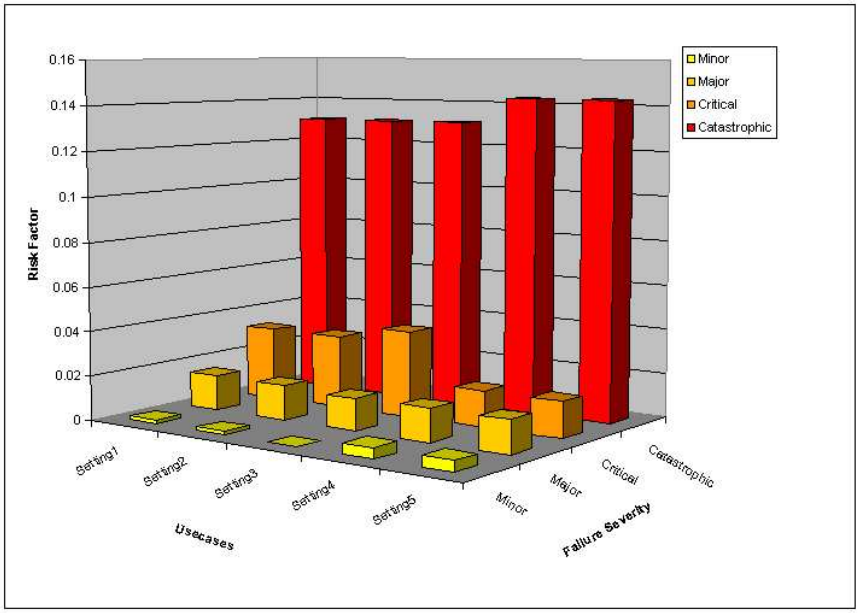

Figure 4.8: The risk distribution of the use cases for the Mode_Setting use case.

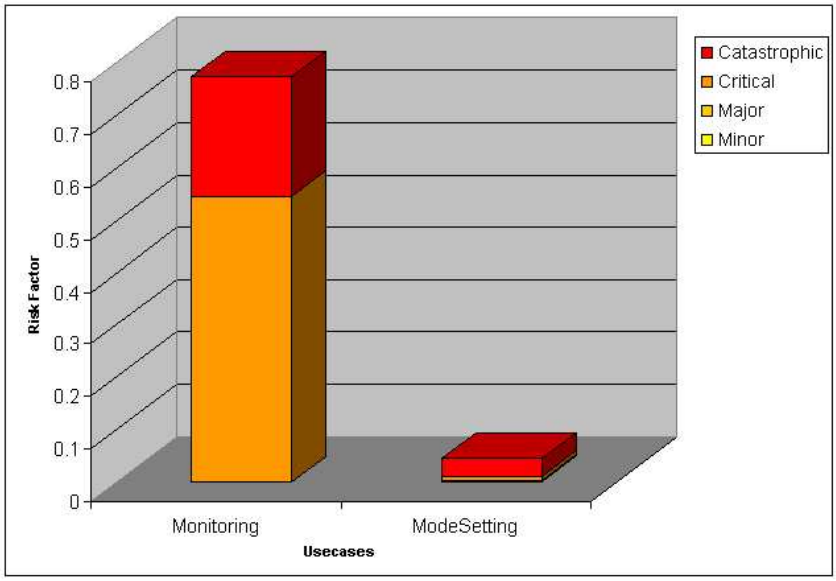

Figure 4.9: The DTMC for the Monitoring use case.

relation ships between the use cases this difference between the use cases could not have been accurate. Moreover the Monitoring use case risk factors fall more into the catastrophic severity class. Hence more attention should be given to the development and testing of monitoring use case as it is a potential source of errors.

Chapter 5

Performance-Based Risk Analysis

This chapter presents performance-based risk analysis. The analysis presented here is also at an architectural-level and based on UML specifications. Performance is a non-functional software attribute that plays an important role in application domains ranging from safety-critical system to e-commerce applications. We introduce the concept of performance-based risk as a risk resulting from the failure of scenarios to meet the specified performance requirements. In particular we are interested in performance attributes such as response time and throughput. Performance-based risk is defined as a combination of the probability to violate a performance objective/requirement and the severity of that violation.

We define performance failure as an unexpected performance result originated from the violation of a non-functional requirement(or objective). Since performance requirements are usually expressed in terms of time, a requirement violation is said to occur when a certain operation takes too long to be completed. This type of failure follows faults that concern system performance rather than system functionalities. For example, the extra time taken from an operation to complete may have been spent in a device that has been saturated due to the system heavy workload. Thus, even though the software system is functionally correct, it may suffer

from performance failures. In some cases with this kind of failure may have worse consequences than functional failures. The performance-based risk analysis is based on annotations of the UML diagrams that support such analysis. The methodology derives the software and hardware parameters from the annotated UML sequence diagrams and deployment diagrams and provides the values for performance-risk for a specified workload.

5.1 Overview of the proposed methodology

In this section we introduce an approach to estimate the performance-risk factor based on failures of a software system modelled with UML diagrams. The methodology mainly identifies risk based on performance requirements that are time-related(e.g. the completion time of a specific operation must be less than a certain threshold). We consider performance failures due to a extended completion time of a certain operation(or if the response time requirement of a particular scenario is not met),for a specifies workload. An operation, i.e. a sequence of actions that a software system performs in order to react to an external trigger, can be described as a UML Sequence Diagram. Therefore our observation point to estimate the probability of a specific performance failure is limited to a sequence diagram. Given the input as annotated UML use case diagrams, sequence diagrams, deployment diagrams and performance objectives for each of scenario we come up with the risk of performance failure of that scenario based for a range of workloads.

The main contribution of this thesis to the performance-risk analysis methodology, presented in [5] are the steps 3 *and* 5. *All the other steps have been described shortly for the sake of completeness.*

The input to our methodology and its steps of are described as follows [5]:

INPUT: Annotated UML diagrams which include Use case diagram, sequence diagrams and deployment diagram; Performance objectives of scenarios(described by sequence diagrams)

1. For each scenario in each use case and for each scenario is that use case;

   (a) *STEP 1* - Assign demand vector to each "action" in sequence diagram and build a Software Execution Model for that scenario

   (b) *STEP 2* - Add hardware platform characteristics on the deployment diagram and conduct stand-alone analysis

   (c) *STEP 3* - Devise the workload parameters; build a System Execution Model and conduct contention-based analysis and estimate probability of performance failure

   (d) *STEP 4* - Conduct severity analysis and estimate severity of performance failure for the scenario

   (e) *STEP 5* - Estimate the performance risk of the scenario; Identify high-risk components

In order to estimate the probability of such a performance failure, we build a model taking into account the sum of completion times of all the actions performed in the scenario. The estimation of the completion time of an operation not only depends on combination of these individual action times, but also the resource contention based on system workload. Thus the response time of a given scenario depends on both the action completion times and the system workload.

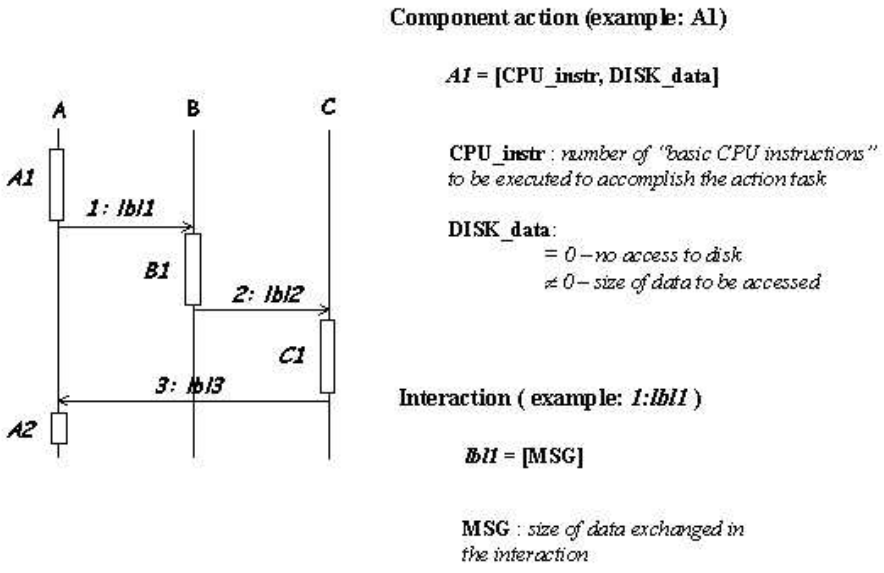

Figure 5.1: The annotated sequence diagram.

5.2 Step 1: Assign demand vector to each "action" in Sequence Diagram and build a Software Execution Model

Figure 5.1 shows the the a sequence diagram, annotated with information related to the resources that each action/interaction needs, in order to be completed. These extensions of UML to represent performance-related concepts have been described in [43] and recently accepted by OMG as a UML profile.

5.2.1 Annotations of the UML Sequence Diagrams

Figure 5.1 shows the annotations that we used in our methodology. We used two kinds of parameters to annotate the account for steps of components and connectors in a sequence diagram. There are again two parameters defined for each action/step of a component, first CPU_{instr}, the number of CPU instructions required to perform this action and second $DISK_{data}$, the number for bytes that are read or written to disk to perform this action. The action/step of connector is identified by one parameter *Interaction* which would contain the size of data that is being transferred across that connector.

The second part of this step translates the sequence diagram(SD) dynamics into a Flow-Graph. After the execution graph is parameterized with demand vectors, it becomes a Execution Graph(EG). This is called the Software Execution Model [36]. Step 2 describes how the Software Execution Model converted System Execution Model by mapping the hardware characteristics described by a deployment diagram.

Similar ideas translation of SD patterns into EG patterns have been given in [37]. A more extensive approach has been introduced in [7], where also asynchronous communication patterns and concurrent action executions have been considered.

5.3 Step 2: Add hardware platform characteristics on the Deployment Diagram; Conduct stand-alone analysis

In order to translate a demand vector in elapsed time, we need to know characteristics of the hardware platform where the software application will be executed. For example, the same number of CPU basic instructions may take, different times depending on the CPU speed or the

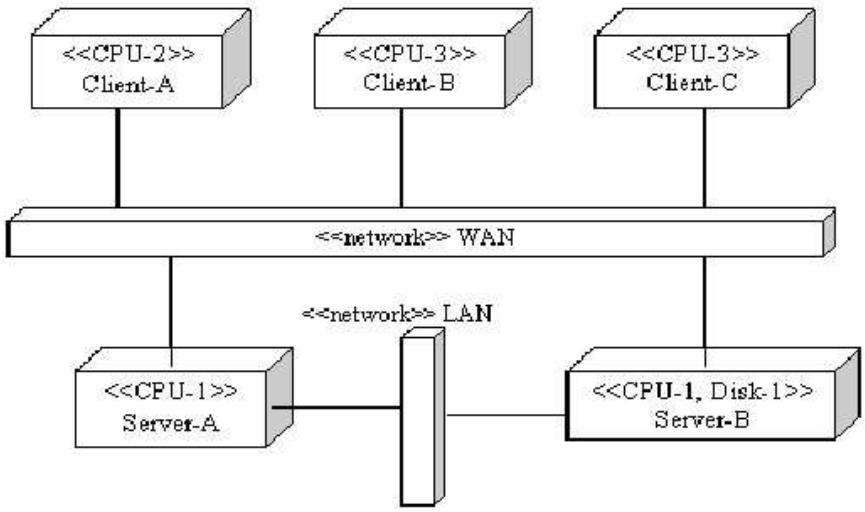

Figure 5.2: The annotated deployment diagram.

same number of disk operations(read/write) may vary depending upon disk speed. In this step we get hardware platform information from an annotated deployment diagram. The advantage of a deployment diagram is that it visually ties the various software components of the system to the hardware components they are executing on.

Figure 5.2 shows an annotated deployment diagram of a hardware platform. Each deployment site can be annotated with the number and type of resources that it hosts. The annotated deployment diagram in figure 5.2 shows two servers, A and B connected by a LAN and the clients A, B and C connected to the servers via a WAN. The site stereotype gives the set of devices allocated on it. Server-A uses a processor of type CPU-1, Server-B uses a processor of type CPU-1, and in addition uses a disk of type Disk-1. Similarly, client A uses a processor of type CPU-2, and clients B and C each use a processor of type CPU-3.

The demand vectors of the software execution model and mapped to the service times of the corresponding hardware and the service demands for each step of the sequence diagram are calculated. Then based on the software execution graph of the scenario, we calculate the

sum of demands (D) for that scenario. While calculating the completion time of a scenario with branching, we consider those branch/path, which have higher service demands and ignore others (this is clearly explained on the case study in section 5.7).

A stand-alone analysis of such a system consists in evaluating the completion time of the whole SD as it would be executed on a dedicate hardware platform with a single user workload. This is an optimistic estimation since it does not consider delays due to contention for resources. Therefore, if the time value from stand-alone analysis violates the performance requirement the failure probability can be considered equal to 1 without any further investigation, and the software system has no feasible implementation under the given set of requirements. Otherwise, it is worth to investigate the system behavior while varying its workload, in order to estimate the failure probability.

5.4 Step 3: Devise the workload parameters;build System Execution Model; conduct contention-based analysis and estimate probability of performance failure

For building the System Execution Model and estimating the probability of performance failure we use the bounding analysis on response time. With bounding analysis it takes very little computation to determine the response time as a function of system workload intensity(population, arrival rate etc.).

There are many advantages of bounding analysis few of which can be listed as follows [25]:

1. The results of these techniques provides a balanced insight into the primary factors affecting

the system performance. The influence of system bottleneck is highlighted and quantified

2. These bounds can be computed simply and quickly (even by hand) and can be used as first cut modelling to eliminate alternatives at an early stage of software life cycle.

3. Bounding analysis can also be used to estimate the potential performance gain of alternatives to existing system.

There are also two types of performance bounds: asymptotic and balanced system bounds. We are interested in asymptotic bounds, which are simpler to estimate, although balances system bounds tend to be more accurate. In order to be able to build System Execution Model and conduct contention-based analysis we first need to define the system workload in one of the following terms: the arrival rate λ (for transaction workloads), or the population N(for batch workloads) or the population N and think time Z(for terminal workloads). In our case we consider batch workloads so the think time is zero.

A complete system contention-based analysis lays on the parametrization of a System Execution Model with values coming from the synthesis of the Software Execution Model mapped via a deployment diagram. The parameterized model can then be solved to obtain performance indices. We are interested in estimating the optimistic(lower) and pessimistic(upper) bounds on system throughput and response time for a given scenario rather than actually solving the performance model. The equations defining the asymptotic bounds on the throughput and response time of a system with batch workload, as a function of number of customers N are given as follows [25]:

Bounds on Throughput: $1/D \leq X(N) \leq \min(N/D, 1/D_{max})$

Bounds on Response Time: $\max(D, N * D_{max}) \leq R(N) \leq N * D$

where N is the number of customers, $D = \sum D_i$ is the sum of all demands in the scenario

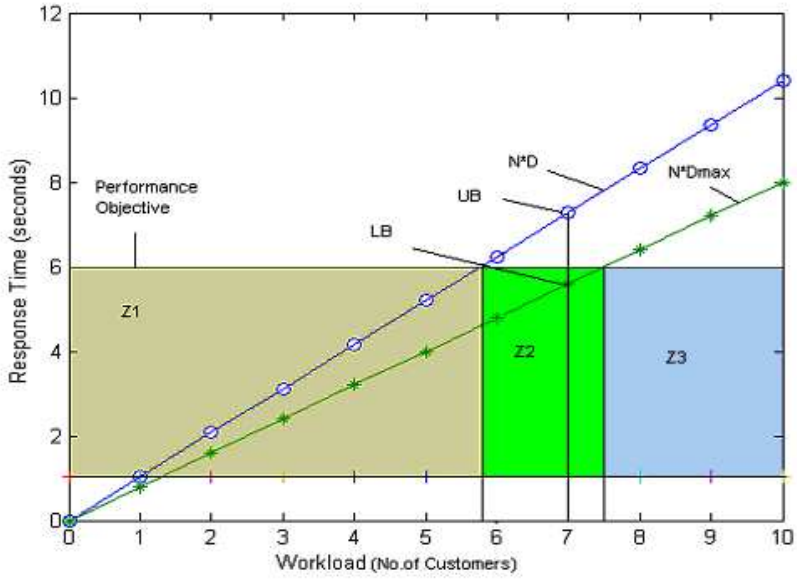

Figure 5.3: The plot showing the asymptotic bounds and failure probability estimates for response time.

and D_{max} is the maximum demand in that scenario.

Figure 5.3 shows a diagram of the asymptotic bounds on response time R(N) versus the workload N (no. of customers). The upper bound $N * D$ is shown as the line marked with (-o-) The lower bound is estimated by comparing D, shown as line marked with (-+-) and $N * D_{max}$, shown as a line marked with (-*-). The values of the actual response time must lie between these three lines. Figure 5.3 shows the a 6 seconds response time objective(parallel to x-axis) and the assumed workload of 7 customers(parallel to y-axis).

1. $Z1$: Failure probability (Z1) = 0.
2. $Z2$: Failure probability (Z2) = $(UB - OBJ)/(UB - LB)$
3. $Z3$: Failure probability (Z3) = 1

   Note: UB stands for Upper Bound, LB for Lower Bound and OBJ stands for Performance Objective.

In order to estimate the probability of a performance failure, we partition the workload domain in three zones. In the $Z1$ zone both upper and lower bound on the response time are below the performance objective, so the probability of failure is zero. Analogously the failure probability holds 1 in the $Z3$ zone, as both bounds fall over the performance objective. In the $Z2$ zone we estimate the failure probability as the ratio between the distance of the upper bound from the performance objective (failure range) and the distance between the bounds (whole range).

5.5 Step 4: Conduct severity analysis and estimate severity of performance failure for the scenario

Severity analysis is performed using Functional Failure Analysis (FFA), based on UML use case Diagrams. FFA is performed on system-level sequence diagrams. This provides a comprehensive view of the ways in which the system can fail, and the severity of the failure. The results of FFA study are recorded in a tabular form. A framework of the severity analysis to come up with the severity values of the scenarios based on performance failures is presented in out previous work [17]. The system-level sequence diagrams show the system states, the actors involved, and the input and output events. Note that the system-level sequence diagram of a scenario is different from the component/connector-interaction sequence diagram. The Internal components and their interactions are not shown in the system-level sequence diagram. The details of coming with the severity values for various scenarios is presented in [5] as is not presented here as it is not a contribution of this thesis.

5.6 Step 5: Estimate the performance risk of the scenario and identify high-risk components

The performance risk of a scenario is defined as the product of two factors:

1. Probability of performance-failure : The probability that the system will fail to meet the required performance objective obtained from STEP 3 5.4
2. Severity of performance-failure : The severity associated with this performance failure of the system in this scenario obtained from STEP 4 5.5.

In addition to estimating performance risk of a scenario (i.e., identifying high-risk scenarios), our methodology also helps in identifying a set of high-risk components that should undergo more rigorous development and implementation and to which should be allocated more testing effort. For this purpose, we first estimate the overall residence time of each component in a given scenario and then normalizing it with the response time of the scenario.

A high-risk component is said to be the one with a higher normalized service time in a certain scenario. In a case of a performance failure in a certain scenario, the component, with a high service time, will be the bottleneck component causing that failure. Moreover a component can be compared across many scenarios and if the service demand for that component in high across many scenarios then that component is certainly critical.

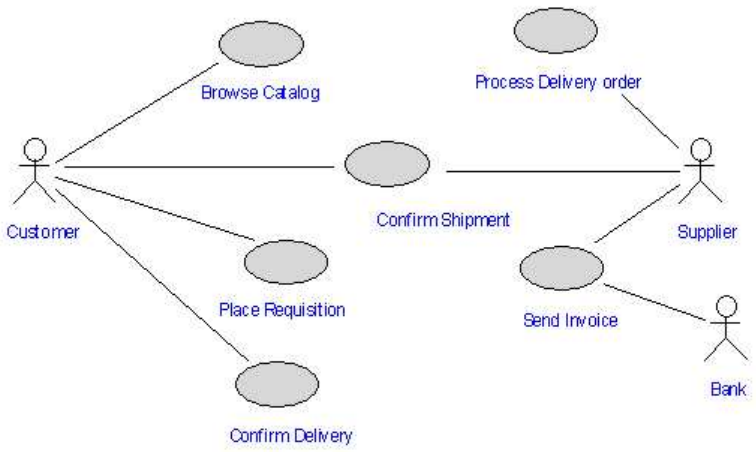

Figure 5.4: The Use case model of the E-commerce case study.

5.7 E-commerce case study

The steps 3 and 5 are the main contributions of this thesis .The application of these steps is explained in detail, while the application of steps 1, 2 and 4 are described shortly for the sake of completeness. For details on applying the steps 1,2 and 4 on the e-commerce case study please refer [5]

In order to validate our risk assessment methodology, we applied it to an e-commerce application. Performance is important for these types of systems, since a slow response may result in the loss of an impatient customer etc. A e-commerce system is a widely used, web-based application, which allows customers and suppliers to interact with each other over the internet through software agents. Briefly describing, the system allows a customer to browse through the various catalogs provided by the suppliers, select the item to be purchased and place the order. The order is validated by checking two things: if the customer has a contract with the supplier; and a bank account/s through which payments can be made. The supplier checks for the availability of the product and if available, ships the product. On receiving the product, the customer sends back an acknowledgement. Finally, the invoice is processed by electronically

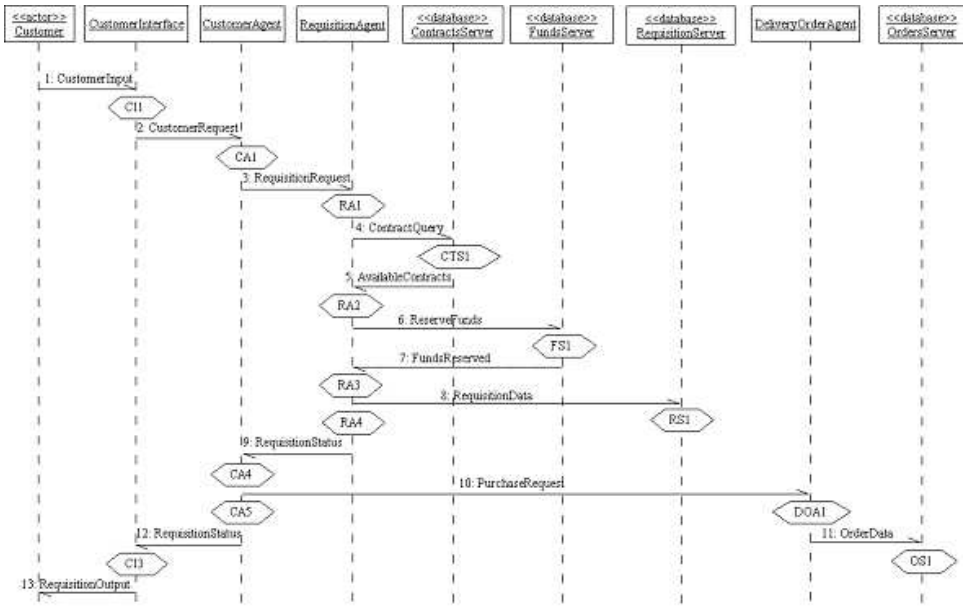

Figure 5.5: The Scenario diagram for Place Requisition scenario.

transferring funds from the customer's bank account to the supplier's. Details of the system are presented in [13].

Figure 5.4 shows the use case model of the e-commerce case study. It consists of six use cases, which are explained in detail in [13]. The Sequence Diagram for place requisition scenario shown in figure 5.5. Applying step 1 on the scenario diagram we, transform it into the Execution Graph shown in figure 5.6. Each rectangle in figure 5.6 is a node representing a process step and the first node(PLACE_REQ) denotes an expanded node. Note that the time taken for the execution of the whole scenario is NOT the sum of time taken by each individual branch, rather, it is the time taken by a single branch or path which takes the longest time when compared to the time taken by the other concurrent branches. We consider only this path for calculating the sum of demands for that scenario. In figure 5.6, the path with the highest demand is shown in bold and it forms the process sequence that takes the longest time and has the highest demand. For details please refer [5].

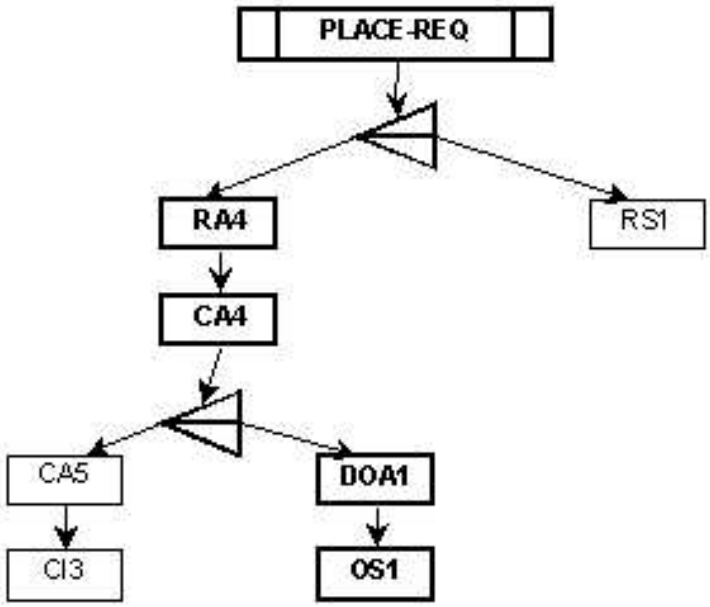

Figure 5.6: The Software Execution Graph of the Place Requisition Scenario.

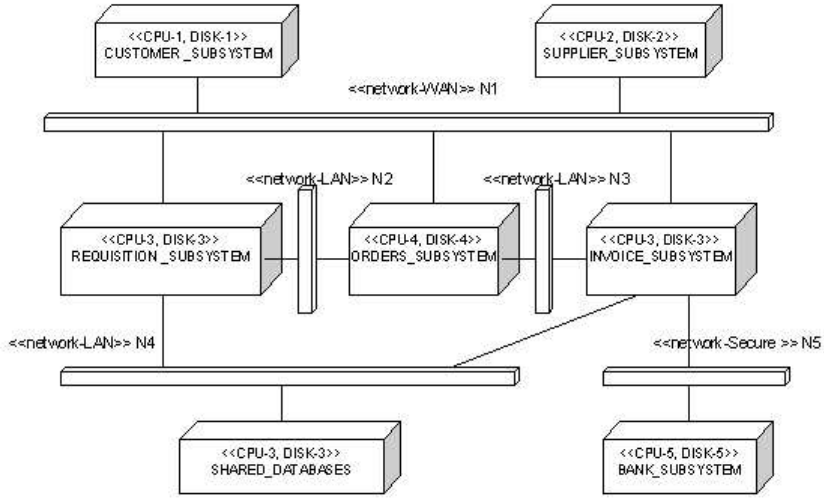

Figure 5.7: The deployment diagram of the E-commerce application.

Table 5.1: The service times of hardware devices

	CPU1	CPU3	CPU4	DISK3	DISK4	LAN	WAN	Local
Units	loc/ns	loc/ns	loc/ns	$\mu sec/KB$	$\mu sec/KB$	$\mu sec/KB$	$\mu sec/KB$	$\mu sec/KB$
Service times	40	10	5	120	60	80	800	5

Figure 5.7 shows the deployment diagram with hardware platform characteristics. Based on the annotations in the deployment diagram we build the service times of the hardware platform devices. There are there CPUs, two Disks, a local area network(LAN) and a wide area network(WAN) identified from the deployment diagram, whose service times are shown in table 5.1. Table 5.2 shows the demand vectors of each step in the place requisition scenario. These are then multiplied with the corresponding values in the table 5.1 to get the values for the system execution model. In the case of the place requisition scenario, the sum of demands (or the completion time of scenario) is equal to 0.7408. For a stand-alone analysis of this scenario (considering the workload of single customer), the performance objective of 6 seconds is well met. Hence we move on to the contention-based analysis and estimate probability of failure for a range of workloads as shown in figure 5.8.

The system execution model for each scenario is built after they pass the stand-alone analysis. Similarly the values of the maximum demand D_{max} and sum of all the demands D is also calculated for all the scenarios. The values for place requisition scenario are D_{max}=0.328 and D=0.7408. We are interested in the risk factor of the scenario with a desired response time of 6 seconds at a workload of 15 customers. Figure 5.8 shows the plot of the asymptotic bounds on the response time on a varying workload for place requisition scenario. All the parameters are labelled in the plot and the values are as follows : The upper and lower bounds for a workload of 15 customers are 11.1120 and 4.9200 respectively and the performance objective for this scenario is 6 seconds. The probability of the performance failure for the scenario is equal to $(UB - OBJ)/(UB - LB) = (11.112 - 6)/(11.112 - 4.92) = 0.8256$.

From the severity analysis described in section 5.5 the severity for the performance failure

Table 5.2: The demand vectors of the Place Requisition scenario

Processing Step	CPU1	CPU3	CPU4	DISK3	DISK4	LAN	WAN	Local
	job	job	job	byte	byte	byte	byte	byte
CI1	1							80
RA1		6				240		
CTS1		2.699		1000		1000		
RA2		4				160		
OFS1		5.699		1000		10		
RA3		6						1000
RS1		2		1000				
RA4		2					10	
CA4	5						240	
DOA1			3					1000
OS1			2		1000			
CA5	2							10
CI3	3							1000

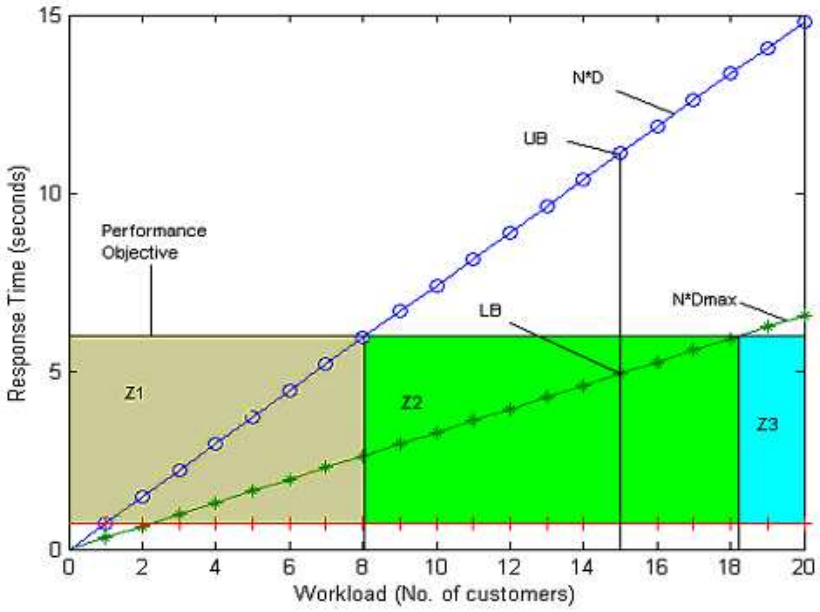

Figure 5.8: The plot showing the asymptotic bounds for response time of place requisition scenario.

of place requisition scenario is catastrophic with associated value 0.95. Hence the risk factor of this scenario is product of both 0.8256 and 0.95 which equals 0.7843.

Table 5.3 shows the probability and severity of performance failures of all the scenarios of the e-commerce case study, along with their risk factors. The place requisition and confirm shipment scenario present high catastrophic risk values and hence need more design and testing efforts. Its worth while to investigate the components in these scenarios.

Figure 5.9 shows the performance-risk factors of the various scenarios in the e-commerce system. The bars are colored according to the four classes of severity. Higher and darker bars are scenarios with high risk factors. In the figure the place requisition scenario and the confirm shipment scenarios are the ones with high risk factors, falling into the catastrophic severity class.

The result of the last step in the methodology i.e identification of the performance-critical

Table 5.3: The performance requirements and the risk factors of the various scenarios in the e-commerce case study

Scenario	Desired response time	Workload	Prob. of failure	Severity of failure	Risk Factor
Browse Catalog	4	4	0.0	0.25	0.0
Place Requisition	6	15	0.8256	0.95	0.7843
Process Delivery order	6	6	0.76	0.5	0.38
Confirm Shipment	6	7	0.7619	0.95	0.7238
Confirm Delivery	2	5	0.9158	0.75	0.6869
Send Invoice	4	7	0.6903	0.75	0.5177

components is shown in Figure 5.10. The components are shown in x-axis the scenarios are shown on y-axis and the normalized-service times of components are shown on z-axis. To obtain these values the sum of service times of the various action-steps of a component in a scenario is normalized against the sum of action-steps of all components in that scenario as shown in section 5.6. It can seen that many components in the confirm shipment and place requisition scenarios have high service demands and moreover the severity assigned to these scenarios is catastrophic, hence these components are the performance-critical and need more investigation. There is also a component in the browse catalog scenario which has a very high service demand value but since the severity assigned to this scenario is only minor this component can be considered as less critical.

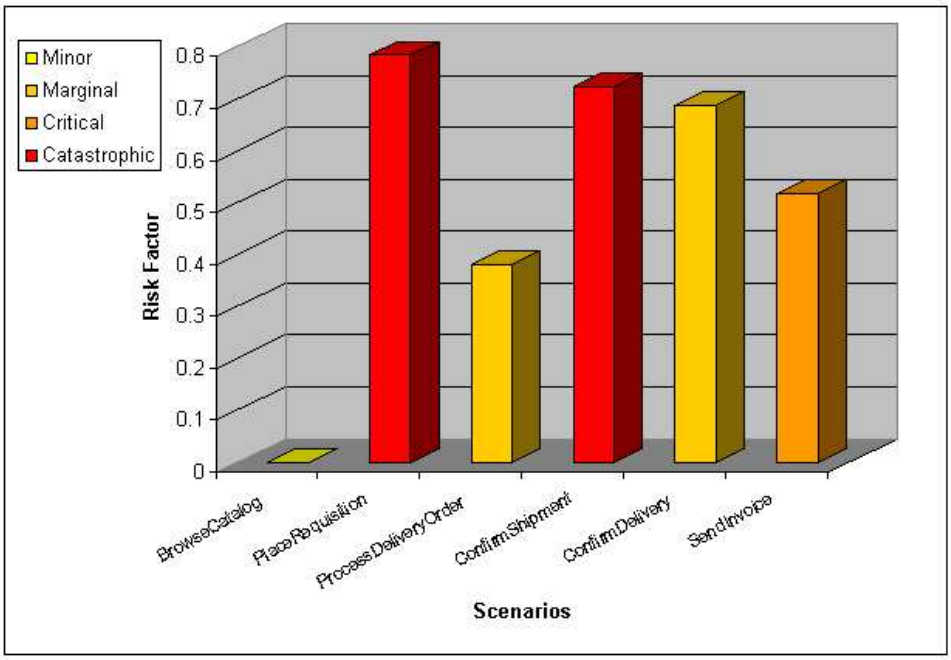

Figure 5.9: The graph showing the performance-risk factor of the various scenarios of e-commerce system.

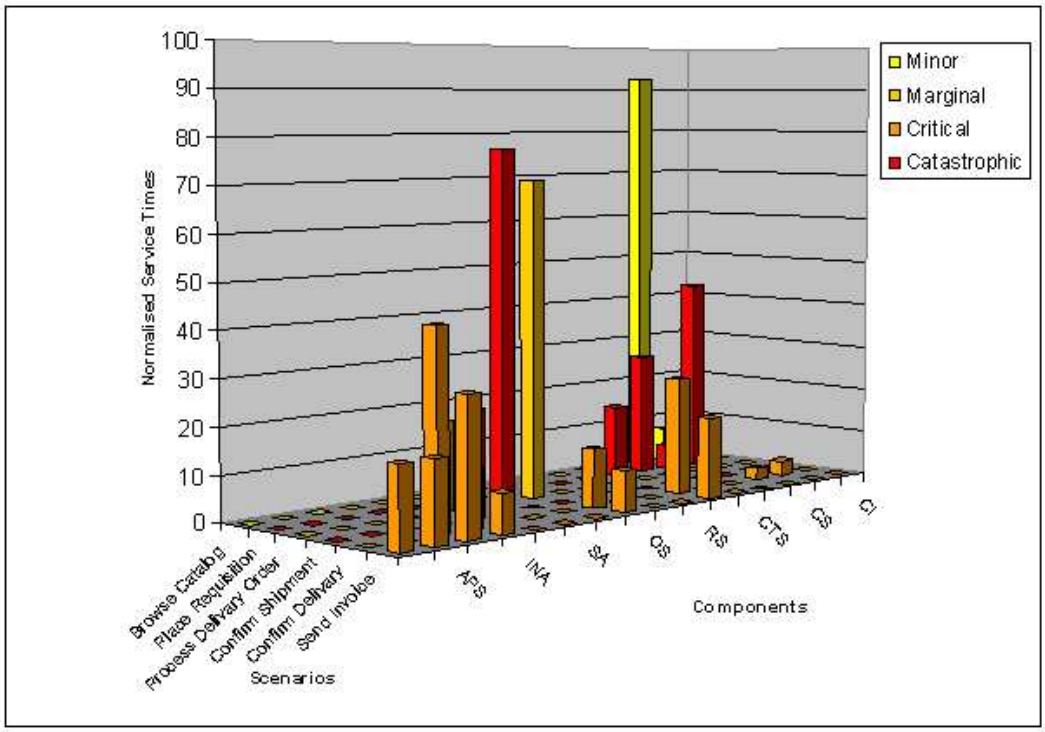

Figure 5.10: The graph showing the performance-critical components of the e-commerce system.

Chapter 6

Conclusions and Future Work

This thesis presents the architectural-level methodologies focussed on assessment of reliability-bjiased and performance-based risk. The reliability-based risk is defined as the product of probability of failure and the severity of the failure. A methodology for calculating the risk factors of various components and connectors and building a risk model from a sequence diagram is presented in chapter 3. This methodology calculates the distribution of scenario risk factor, among the various severity classes by solving the Markov chain of the risk model. Then it aggregates the risk factors of scenarios to give the use case risk factors. The system-level risk factor is calculated by averaging the use case risk factors with their execution probabilities. The risk methodology is applied on the cardiac pacemaker case study in section 3.7. Since the methodology is entirely analytical and provides a closed form solution, it is very suitable for sensitivity analysis and automation. In fact, a prototype of the risk assessment tool written in JAVA which reads the embedded UML information from Rational Rose, and calculates the various risk factors has already been developed.

An extension to this methodology, which relaxes the assumption of independent use cases i.e considers the various relationships between the use cases is presented in chapter 4. The

algorithm for automation of the risk aggregations process, presented in section 4.1.4 works in two passes. In the first pass the algorithm scans the entire use case diagram in a depth-first manner and colors the use cases according to the relationships. It colors the primitive, non-primitive and the terminal usecases each with a different color and also calculates the primitive use case risk factors. In the second pass the algorithm works in a bottom-up fashion. It uses the coloring scheme, checks if risk factors of all the use cases directly related to a non-primitive use case are calculated and then calculates the risk factor of that non-primitive use case. This is repeated until all the non-primitive and the terminal use case risk factors are calculated. Finally the algorithm computes the system-level risk factor based in the specified execution probabilities of the terminal use cases. This algorithm is applied on the HCS case study in section 4.2. As part of the future work this algorithm should be integrated in the risk assessment tool.

In performance-based risk analysis presented in the chapter 5 defines performance-risk as the product of the probability of a performance failure and the severity of the performance failure. The performance failures of the each scenario is analyzed by building a software execution model of that scenario with the demand vectors. The software execution is converted to a system execution model based in the deployment diagram information about the hardware platform. A stand alone analysis is conducted for each scenario and the if the scenario passes the stand alone performance requirements a contention-based analysis is conducted, using the asymptotic bounding analysis. This kind of analysis is conducted at a scenario level and the risk factors are calculated from the bounds on the response time for a suitable workload for that scenario. As part of the future work for performance-based risk analysis we are looking at bounds on throughput for the scenarios along with the response time. A balanced system bounding analysis can also be conducted, which provides a higher accuracy than the asymptotic bounds.

Bibliography

[1] H. Ammar, T. Nikzadeh, and J. Dugan, "A Methodology for Risk Assessment of Functional Specification of Software Systems Using Coherent Petri Nets", Proc. Fourth Intl Software Metrics Symp. (Metrics 97), pp. 108-117, 1997.

[2] J. Bowles, "The New SEA FMECA Standard", Proc. Ann. Reliability and Maintainability Symposium(RAMS 1998), pp. 48-53, 1998.

[3] R.C. Cheung, "A User-Oriented Software Reliability Model", IEEE Trans. Software Eng., vol. 6, no. 2, pp. 118-125, 1980.

[4] Cortellessa V. and Mirandola R., PRIMA-UML: "A Performance Validation Incremental Methodology on Early UML Diagrams, Science of Computer Programming", Elsevier Science, vol.44, no.1, pp.101-129, July 2002.

[5] V. Cortellessa1, K. Goseva-Popstojanova, K. Appukutty, A. Guedem, A. Hassan, R. Elnaggar, W. Abdelmoez, and H. H. Ammar "Performance-based Risk Analysis of UML Models" (to be submitted)

[6] *Procedures for Performing Failure Mode Effects and Criticality Analysis*, US MIL_STD_1629 Nov. 1974, US MIL_STD_1629A Nov. 1980, US MIL_STD_1629A/Notice 2, Nov. 1984.

[7] Di Berardino A., "Design of an algorithm to translate annotated UML Sequence Diagrams into Execution Graphs", from the Master Thesis "Experimenting software risk analysis" (in italian), University of L'Aquila, April 2003.

[8] Evgeni Dimitrov, Andreas Schmietendorf and Reiner Dumke "UML-Based Performance Engineering Possibilities and Techniques", IEEE Software, Jan-Feb,2002.

[9] Douglass B. P., "Real Time UML: Developing Efficient Objects for Embedded Systems", addison Wesley, 2nd Edition 2000.

[10] K. El Emam and W. Melo, "The Prediction of Faulty Classes Using Object-Oriented Design Metrics", Technical Report NRC 43609, Natl Research Council Canada, Inst. for Information Technology, 1999.

[11] N. Fenton and N. Ohlsson, Quantitative Analysis of Faults and Failures in a Complex Software System, IEEE Trans. Software Eng., vol. 26, no. 8, pp. 797-814, Aug. 2000.

[12] Gomaa H., *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-Wesley, 2000.

[13] Gomaa H. and Menasce D.A., "Design and Performance Modeling of Component Interconnection Patterns for Distributed Software Architecture", Proc. of Second International Workshop on Software and Performance, WOSP2000, September 2000, Ottawa, Canada, 2000, pp.117-126

[14] K. Goseva-Popstojanova and K.S. Trivedi, "Architecture Based Approach to Reliability Assessment of Software Systems", Performance Evaluation, vol. 45, nos. 2-3, pp. 179-204, June 2001.

[15] K. Goseva-Popstojanova , A. Hassan, A. Guedem, W. Abdelmoez, D. Nassar, H. Ammar, A. Mili, "Architectural-Level Risk Analysis using UML", IEEE Trans. Software Engineering, Vol. 29, No.10, October 2003, pp. 946-960 .

[16] Hassan A., Goseva-Popstojanova K., Ammar H., "Methodology for Architecture Level Hazard Analysis: A Survey", ACS/IEEE International Conference on Computer Systems and Applications, (AICCSA'03), Tunis, Tunisia, July 14-18, 2003

[17] Hassan A., Abdelmoez W., Guedem A., Apputkutty K., Goseva-Popstojanova K., Ammar H., "Severity Analysis at Architectural Level Based on UML Diagrams", 21st International system Safety Conference, Ottawa, Canda, August 4th -8th, 2003, pp.571-580.

[18] W. Harrison, "Using Software Metrics to Allocate Testing Resources", J. Management Information Systems, vol. 4, no. 4, pp. 93-105, 1988.

[19] D. Heimann, "Using Complexity Tracking in Software Development", Proc. Ann. Reliability and Maintainability Symp. (RAMS 1995), pp. 433-437, 1995

[20] Kahkipuro P., "UML based Performance Modeling Framework for Object-Oriented Distributed Systems", Proc. of Second International Conference on the Unified Modeling Language, October 28-30, 1999, LNCS, Springer Verlag, vol.1723, 1999,pp. 356-371.

[21] Jain R., *Art of Computer Systems Performance Analysis*, New York Wiley, 1990.

[22] T. Khoshgoftaar and J. Munson, "Predicting Software Development Errors Using Software Complexity Metrics, Proc. Software Reliability and Testing, pp. 20-28, 1995.

[23] T. Khoshgoftaar, J. Munson, and D. Lanning, "Dynamic System Complexity, Proc. Intl Software Metrics Symp. (Metrics 93) pp. 129- 140, May 1993.

[24] T. Khoshgoftaar, E. Allen, K. Kalaichelvan, and N. Goel, The Impact of Software Evolution and Reuse on Software Quality, Empirical Software Eng., vol. 1, pp. 31-44, 1996.

[25] Lazowska, E., *Quantitative System Performance, Computer System Analysis Using Queuing Network Models*, Prentice Hall, 1984.

[26] Merseguer J., Campos J. and Mena E., "A Pattern-Based Approach to Model Software Performance", Proc. of Second International Workshop on Software and Performance, WOSP2000, September 2000, Ottawa, Canada, 2000, pp.137-142

[27] Merseguer J., Campos J. and Mena E., "Performance Evaluation for the design of Agent-based Systems: A Petri Net Approach", Proc. of Software Engineering and Petri Nets (SEPN 2000), June 2000, Aarhus, Denmark, 2000,pp. 1-20

[28] J. Munson and T. Khoshgoftaar, "Sotware Metrics for Reliability Assessment, Handbook of Software Reliability Eng., M. Lyu, ed., pp. 493-529, 1996.

[29] J. Musa, G. Fuoco, N. Irving, D. Kropfl, and B. Juhlin, "The Operational Profile, Handbook of Software Reliability Eng., M. Lyu, ed., pp. 167-216, 1996.

[30] *NASA Technical Std. NASA-STD-8719.13A, Software Safety*, 1997.

[31] *NASA Safety Manual NPG 8715.3*, Jan. 2000.

[32] Pooley R. and Kabajunga C., "Simulation of UML Sequence Diagrams" Proc. of 14th UK Performance Engineering Workshop, Edinburgh, R. Pooley and N. Thomas Eds., UK PEW '98, July 1998.

[33] Pumfrey D. J., "The Principled Design of Computer System Safety Analyses", Ph.D thesis, University of York, Department of Computer Science, September 1999.

[34] C.V.Ramanamoorthy. F.B.Bastani, "Software reliability- status and perspectives"IEEE Trans. Software Engineering , 8(4) 1982 354-371.

[35] Rational Rose Real-Time, http://www.rational.com/products/rosert/index.jtmpl,2003.

[36] Smith, C.U., *Performance Engineering of Software Systems*, SEI Series in Software Engineering, Addison-Wesley, Readings, Mass,1990.

[37] Smith, C.U., Williams L.G., *Performance Solutions: A Practical Guide To Creating Responsive, Scalable Software*, Addison-Wesley, 2002.

[38] C. Sundararajan, *Guide to Reliability Engineering, Data, Analysis, Applications, Implementation and Management*, Van Nostrand Reinhold, New York, 1991.

[39] S. Yacoub, T. Robinson, and H. Ammar, "A Matrix-Based Approach to Measure Coupling in Object-Oriented Designs", J. Object Oriented Programming, vol. 13, no. 7, pp. 8-19, Nov. 2000.

[40] S. Yacoub, H. Ammar, and T. Robinson, "Dynamic Metrics for Object-Oriented Designs", Proc. Sixth Intl Symp. Software Metrics(Metrics 99), pp 50-61, 1999.

[41] S. Yacoub and H. Ammar, "A Methodology for Architectural- Level Reliability Risk Analysis", IEEE Trans. Software Eng, vol. 28, no. 6, pp. 529-547, June 2002.

[42] *Unified Modelling Language Conference*, 1998, 1999, 2000, 2001, 2002.

[43] *UML Profile for Schedulability, Performance and Time*, ptc/02-03-02, OMG Adopted Specification, http://www.omg.org.

[44] T. Wang, A. Hassan, A. Guedem, W. Abdelmoez, K. Goseva- Popstojanova, and H. Ammar, "Architectural Level Risk Assessment Tool Based on UML Specification", Proc. Intl Conf. Software Eng. (ICSE 2003), pp. 808-809, May 2003.

[45] W. Abdelmoez, A. Guedem, A. Hassan, K. Goseva-Popstojanova, H. Ammar "Architectural-Level Risk Analysis With Use Case Dependencies" (to be submitted)

[46] *M.Xie Software Reliability Modelling*, World Scientific, Singapore 1991.